

# Design of Digital Circuits

## Lecture 21: Graphics Processing Units

Prof. Onur Mutlu

ETH Zurich

Spring 2019

10 May 2019

# We Are **Almost** Done With This...

---

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
- Other Execution Paradigms

# Approaches to (Instruction-Level) Concurrency

---

- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- Superscalar Execution
- VLIW
- Fine-Grained Multithreading
- Systolic Arrays
- Decoupled Access Execute
- SIMD Processing (Vector and array processors, GPUs)

# Readings for this Week

---

## ■ Required

- Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro 2008.

## ■ Recommended

- Peleg and Weiser, "MMX Technology Extension to the Intel Architecture," IEEE Micro 1996.



# SIMD Processing: Exploiting Regular (Data) Parallelism

# Recall: Flynn's Taxonomy of Computers

---

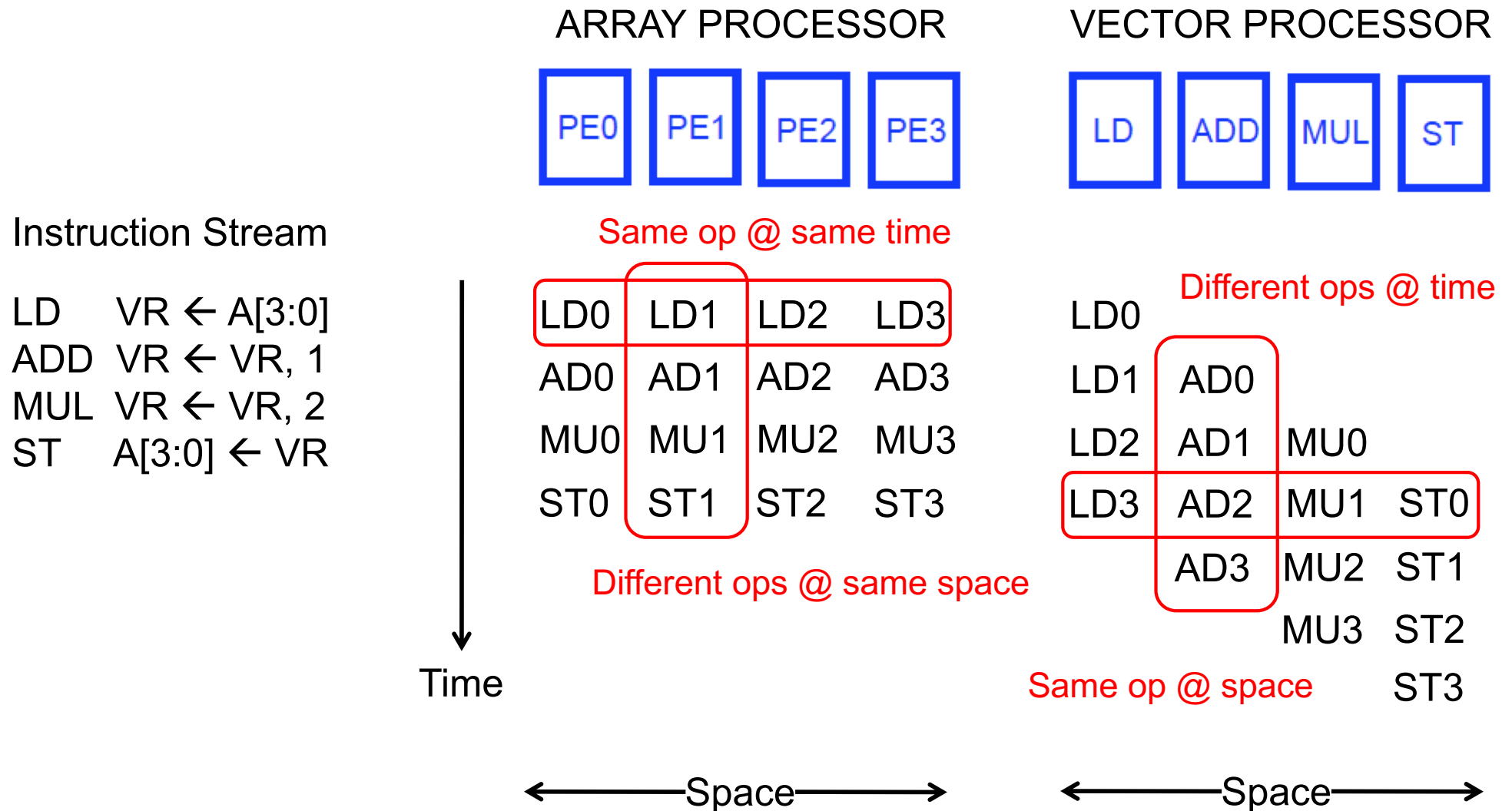
- Mike Flynn, “**Very High-Speed Computing Systems**,” Proc. of IEEE, 1966
- **SISD**: Single instruction operates on single data element
- **SIMD**: Single instruction operates on multiple data elements
  - Array processor
  - Vector processor
- **MISD**: Multiple instructions operate on single data element
  - Closest form: systolic array processor, streaming processor
- **MIMD**: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor

# Recall: SIMD Processing

---

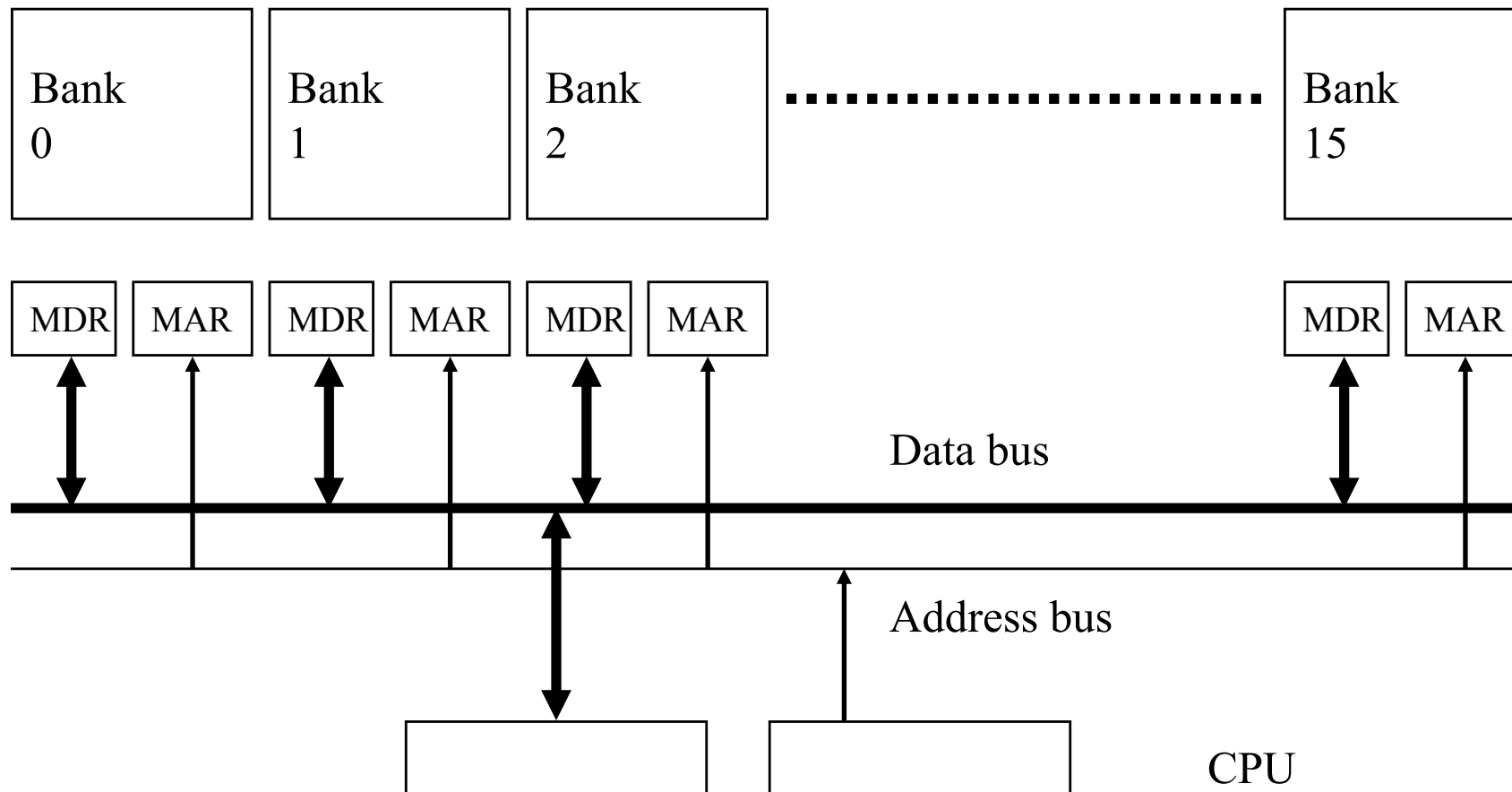
- Single instruction operates on multiple data elements
  - In time or in space
- Multiple processing elements
- Time-space duality
  - **Array processor**: Instruction operates on multiple data elements at the **same time** using **different spaces**
  - **Vector processor**: Instruction operates on multiple data elements in **consecutive time steps** using the **same space**

# Recall: Array vs. Vector Processors

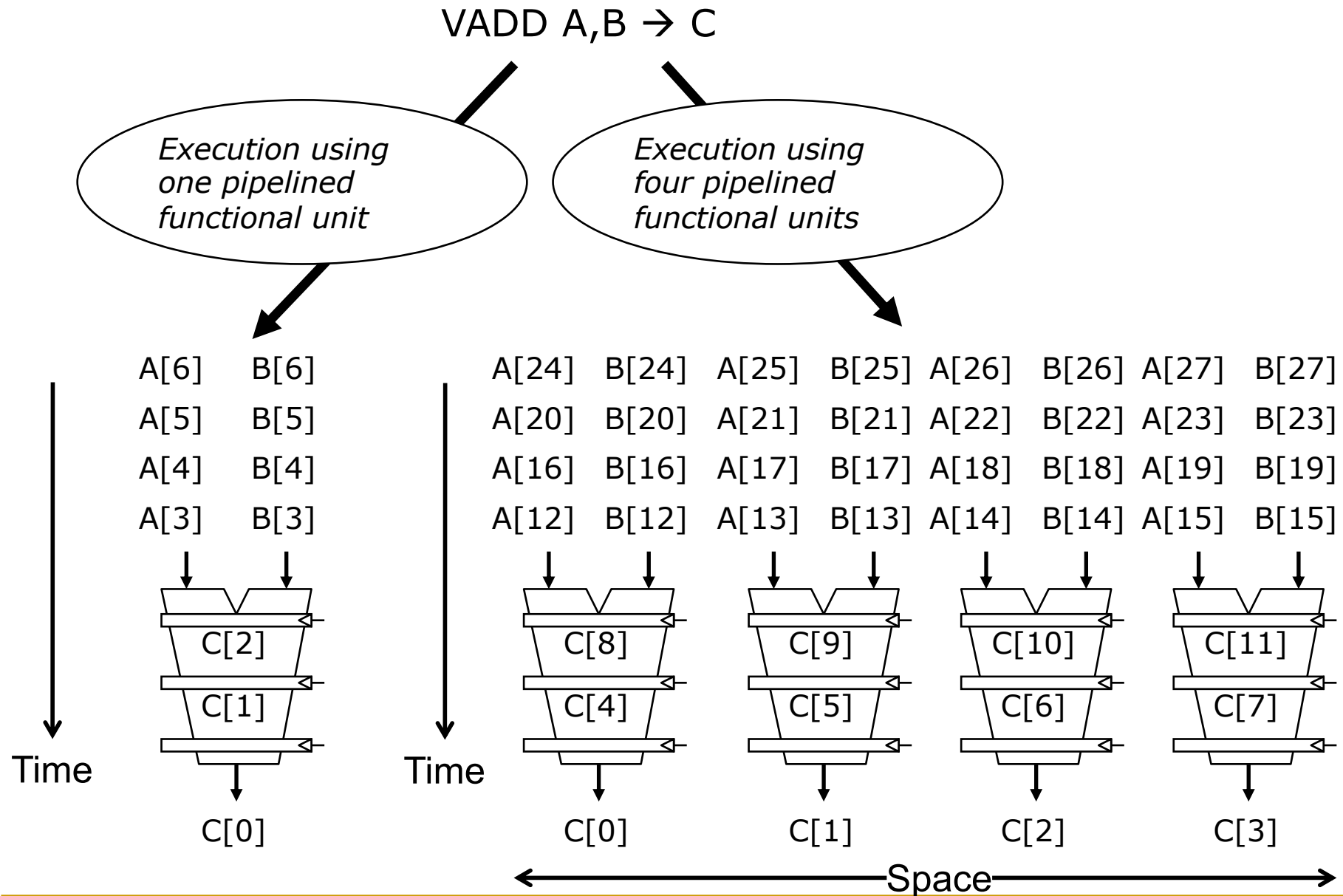


# Recall: Memory Banking

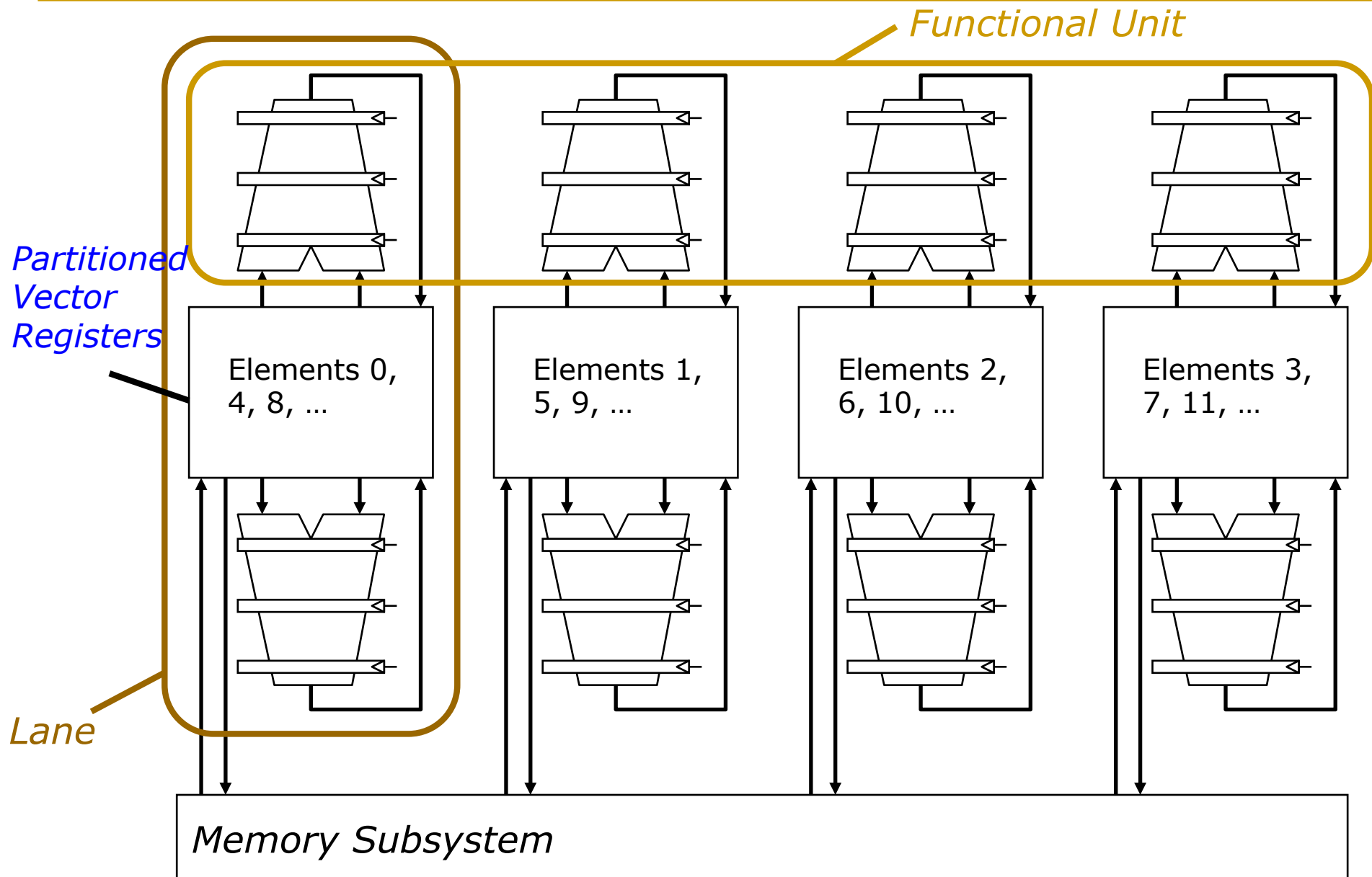
- Memory is divided into **banks** that can be accessed independently; banks share address and data buses (to minimize pin cost)
- Can start and complete one bank access per cycle
- Can sustain N parallel accesses if all N go to different banks



# Recall: Vector Instruction Execution



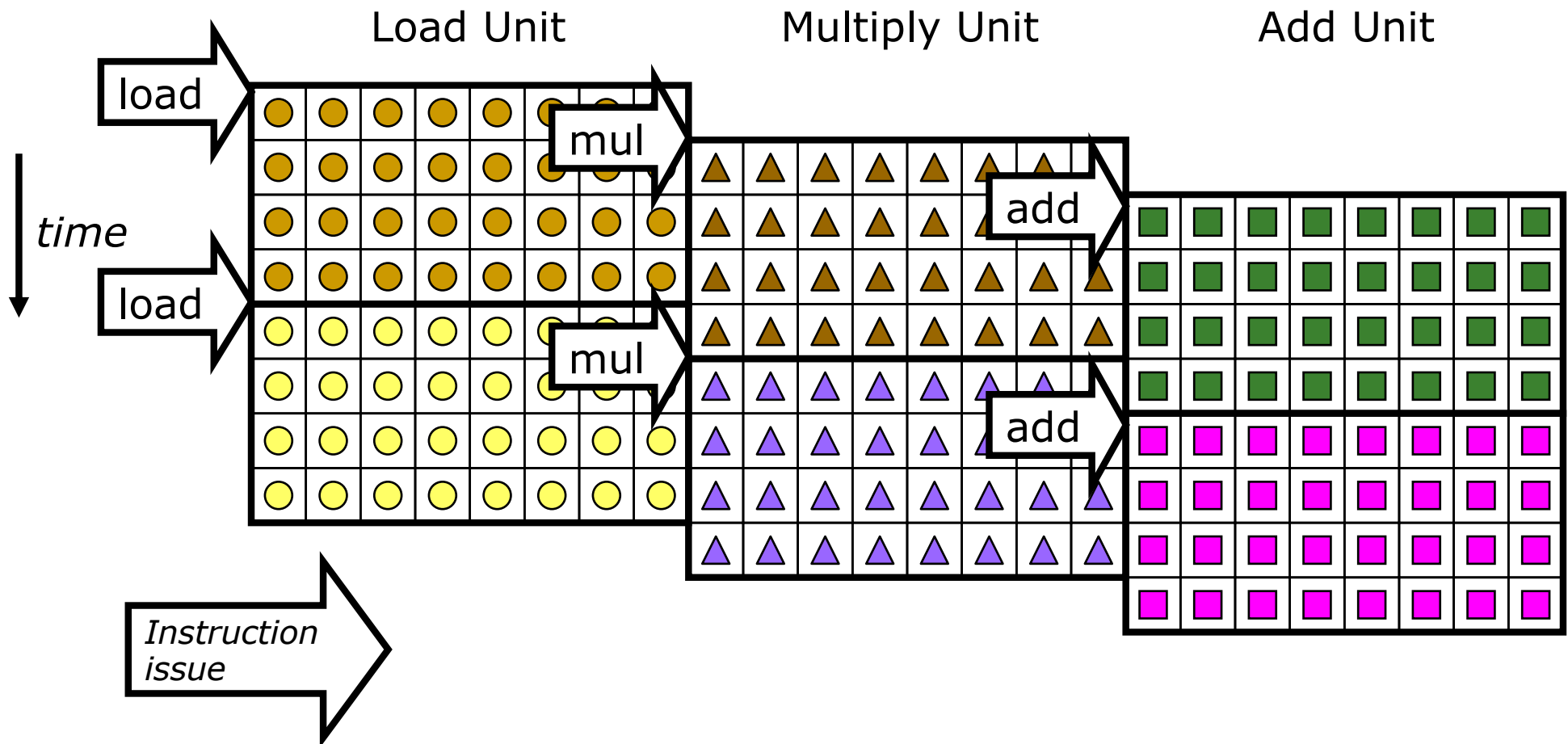
# Recall: Vector Unit Structure



# Recall: Vector Instruction Level Parallelism

Can overlap execution of multiple vector instructions

- Example machine has 32 elements per vector register and 8 lanes
- Completes 24 operations/cycle while issuing 1 vector instruction/cycle

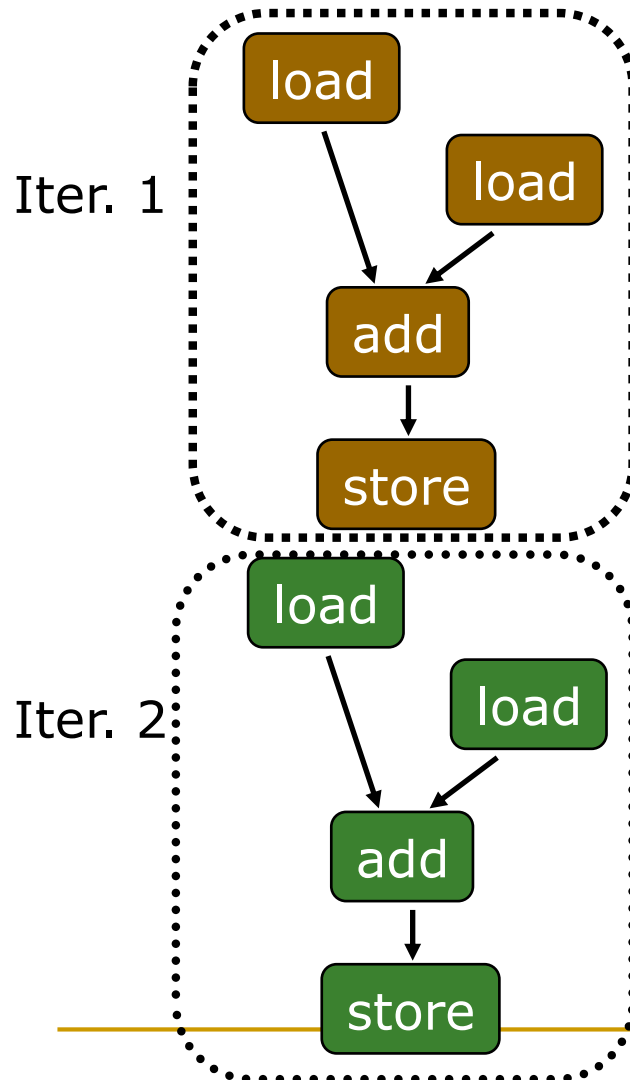




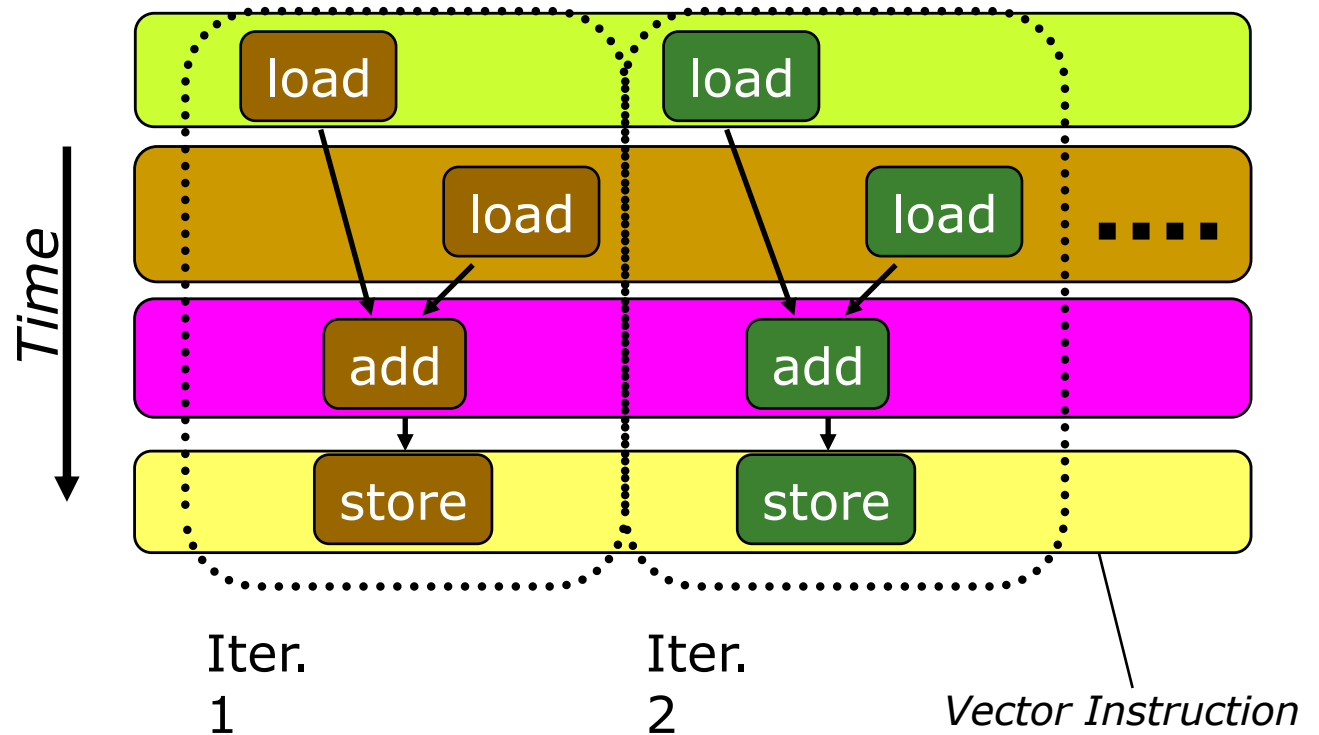
# Automatic Code Vectorization

```
for (i=0; i < N; i++)  
  C[i] = A[i] + B[i];
```

*Scalar Sequential Code*



*Vectorized Code*



Vectorization is a compile-time reordering of operation sequencing  
⇒ requires extensive loop dependence analysis

# Vector/SIMD Processing Summary

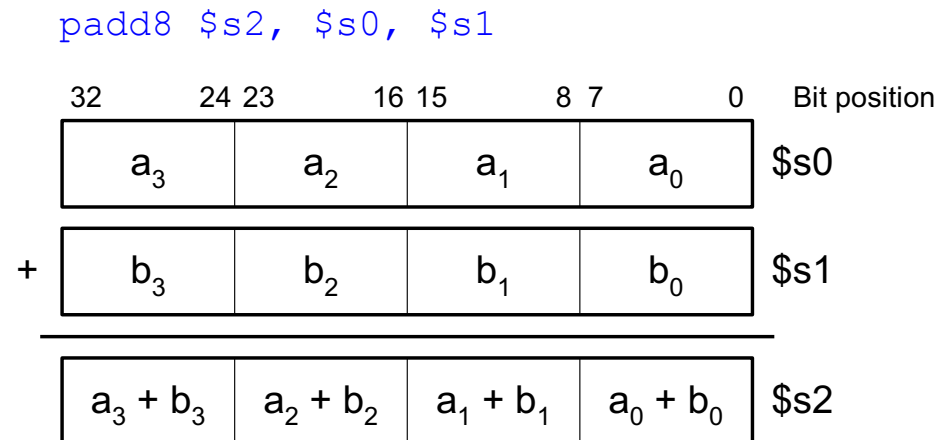
---

- Vector/SIMD machines are good at exploiting **regular data-level parallelism**
  - Same operation performed on many data elements
  - Improve performance, simplify design (no intra-vector dependencies)
- **Performance improvement limited by vectorizability** of code
  - Scalar operations limit vector machine performance
  - Remember **Amdahl's Law**
  - CRAY-1 was the fastest SCALAR machine at its time!
- Many existing ISAs include (vector-like) SIMD operations
  - Intel MMX/SSEn/AVX, PowerPC AltiVec, ARM Advanced SIMD

# SIMD Operations in Modern ISAs

# SIMD ISA Extensions

- Single Instruction Multiple Data (SIMD) extension instructions
  - Single instruction acts on multiple pieces of data at once
  - Common application: graphics
  - Perform short arithmetic operations (also called *packed arithmetic*)
- For example: add four 8-bit numbers
- Must modify ALU to eliminate carries between 8-bit values



# Intel Pentium MMX Operations

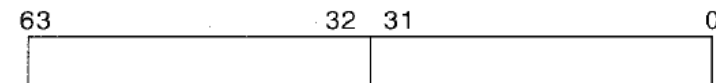
- Idea: One instruction operates on multiple data elements **simultaneously**
  - *À la* array processing (yet much more limited)
  - Designed with multimedia (graphics) operations in mind



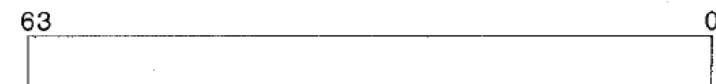
(a)



(b)



(c)



(d)

Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).

No VLEN register

**Opcode** determines data type:

8 8-bit bytes

4 16-bit words

2 32-bit doublewords

1 64-bit quadword

**Stride** is always equal to 1.

Peleg and Weiser, “**MMX Technology Extension to the Intel Architecture**,”  
IEEE Micro, 1996.

# MMX Example: Image Overlaying (I)

- Goal: Overlay the human in image 1 on top of the background in image 2



Figure 8. Chroma keying: image overlay using a background color.

```
for (i=0; i<image_size; i++) {
    if (x[i] == Blue) new_image[i] =y[i];
    else new_image[i] = x[i];
}
```

PCMPEQB MM1, MM3

MM1	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue
MM3	X7!=blue	X6!=blue	X5=blue	X4=blue	X3!=blue	X2!=blue	X1=blue	X0=blue
MM1	0x0000	0x0000	0xFFFF	0xFFFF	0x0000	0x0000	0xFFFF	0xFFFF



Bitmask

Figure 9. Generating the selection bit mask.

# MMX Example: Image Overlaying (II)

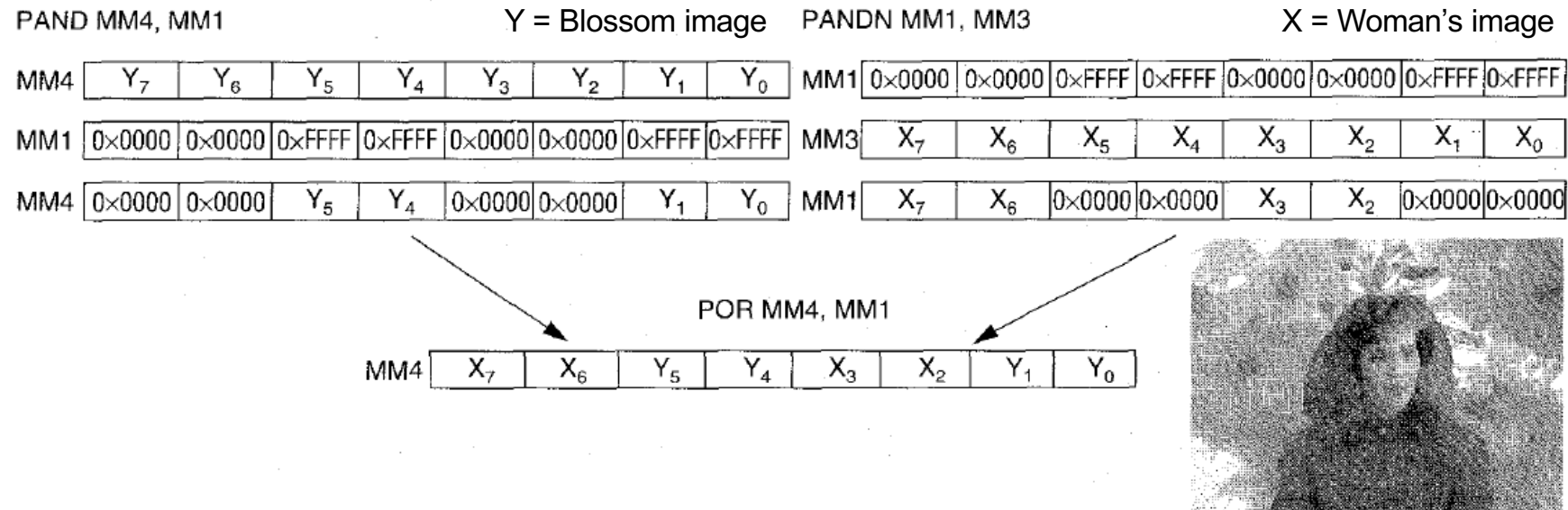


Figure 10. Using the mask with logical MMX instructions to perform a conditional select.

```

Movq    mm3, mem1    /* Load eight pixels from
                       woman's image
Movq    mm4, mem2    /* Load eight pixels from the
                       blossom image

Pcmpeqb mm1, mm3
Pand    mm4, mm1
Pandn   mm1, mm3
Por     mm4, mm1
    
```

Figure 11. MMX code sequence for performing a conditional select.

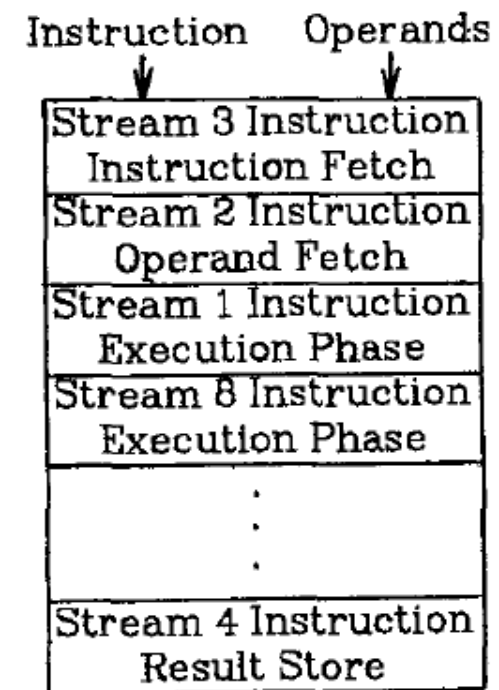
# Fine-Grained Multithreading



# Recall: Fine-Grained Multithreading

- Idea: Hardware has multiple thread contexts (PC+registers). Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread
  - Branch/instruction resolution latency overlapped with execution of other threads' instructions

- + No logic needed for handling control and data dependences within a thread
- Single thread performance suffers
- Extra logic for keeping thread contexts
- Does not overlap latency if not enough threads to cover the whole pipeline

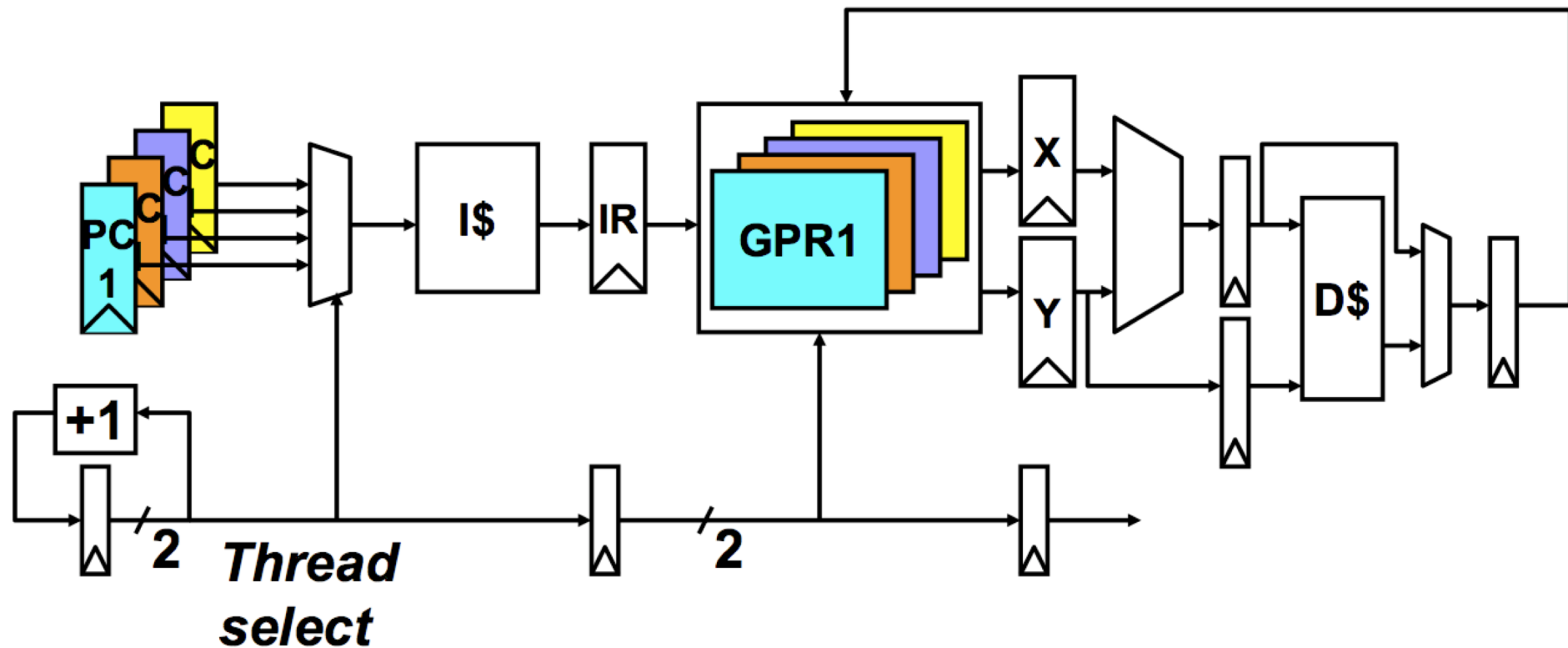


# Recall: Fine-Grained Multithreading (II)

---

- Idea: Switch to another thread every cycle such that no two instructions from a thread are in the pipeline concurrently
- Tolerates the control and data dependency latencies by overlapping the latency with useful work from other threads
- Improves pipeline utilization by taking advantage of multiple threads
- Thornton, “Parallel Operation in the Control Data 6600,” AFIPS 1964.
- Smith, “A pipelined, shared resource MIMD computer,” ICPP 1978.

# Recall: Multithreaded Pipeline Example



# Recall: Fine-grained Multithreading

---

## ■ Advantages

- + No need for dependency checking between instructions  
(only one instruction in pipeline from a single thread)
- + No need for branch prediction logic
- + Otherwise-bubble cycles used for executing useful instructions from different threads
- + Improved system throughput, latency tolerance, utilization

## ■ Disadvantages

- Extra hardware complexity: multiple hardware contexts (PCs, register files, ...), thread selection logic
- Reduced single thread performance (one instruction fetched every N cycles from the same thread)
- Resource contention between threads in caches and memory
- Some dependency checking logic *between* threads remains (load/store)

# GPUs (Graphics Processing Units)

# GPUs are SIMD Engines Underneath

---

- The **instruction pipeline** operates like a **SIMD pipeline** (e.g., an array processor)
- However, the **programming is done using threads**, NOT SIMD instructions
- To understand this, let's go back to our parallelizable code example
- But, before that, let's distinguish between
  - **Programming Model (Software)**
  - vs.
  - **Execution Model (Hardware)**

# Programming Model vs. Hardware Execution Model

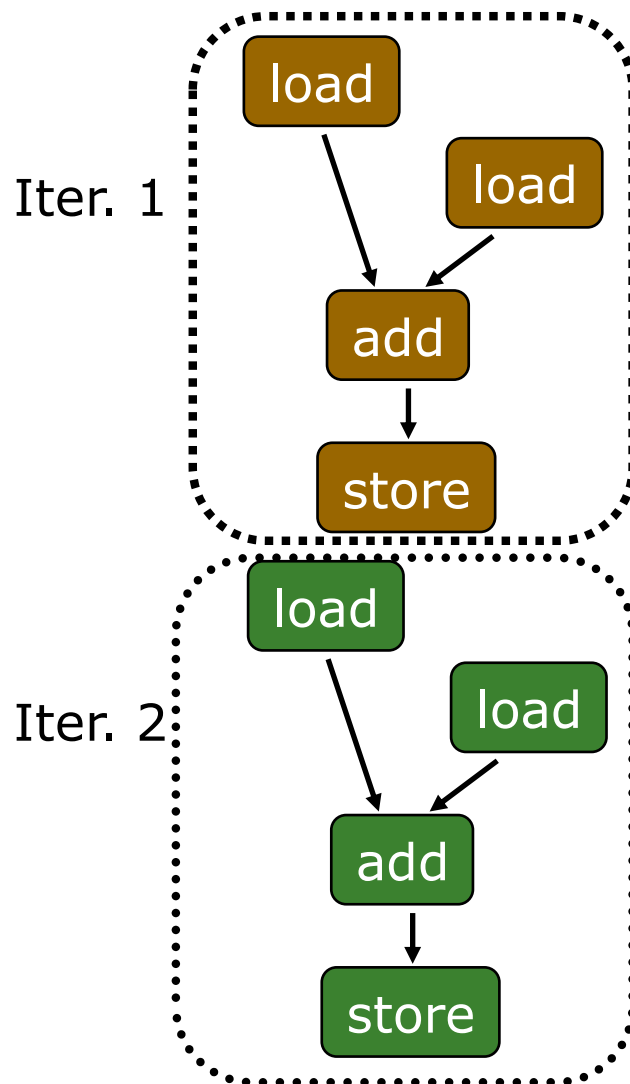
---

- Programming Model refers to **how the programmer expresses the code**
  - E.g., Sequential (von Neumann), Data Parallel (SIMD), Dataflow, Multi-threaded (MIMD, SPMD), ...
- Execution Model refers to **how the hardware executes the code underneath**
  - E.g., Out-of-order execution, Vector processor, Array processor, Dataflow processor, Multiprocessor, Multithreaded processor, ...
- **Execution Model can be very different from the Programming Model**
  - E.g., von Neumann model implemented by an OoO processor
  - E.g., SPMD model implemented by a SIMD processor (a GPU)

# How Can You Exploit Parallelism Here?

```
for (i=0; i < N; i++)  
  C[i] = A[i] + B[i];
```

*Scalar Sequential Code*



Let's examine three programming options to exploit instruction-level parallelism present in this sequential code:

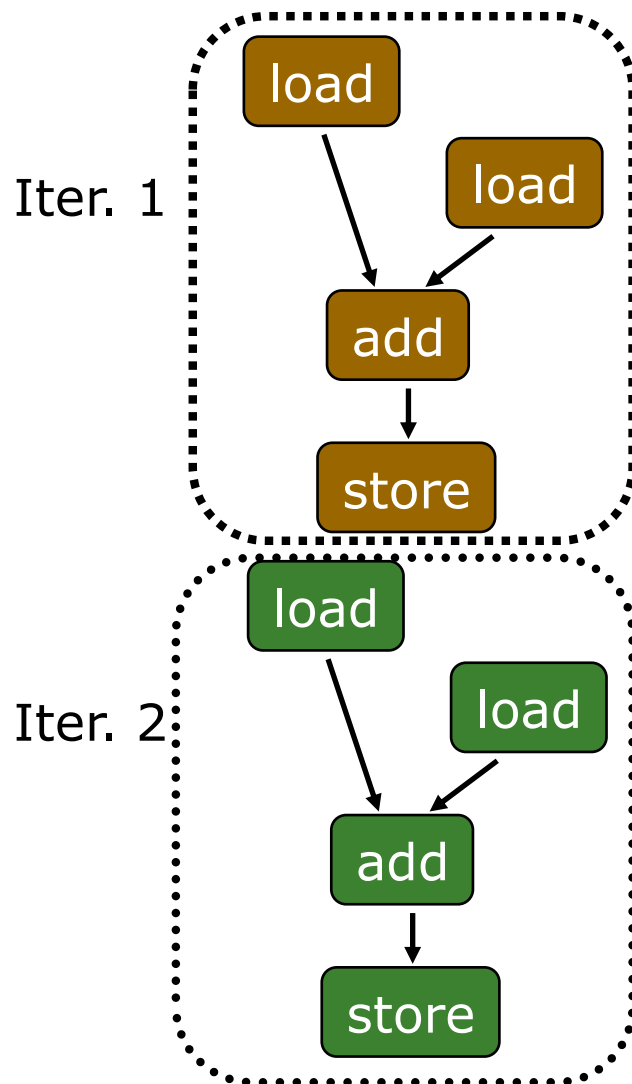
1. Sequential (SISD)
2. Data-Parallel (SIMD)
3. Multithreaded (MIMD/SPMD)



# Prog. Model 1: Sequential (SISD)

```
for (i=0; i < N; i++)  
    C[i] = A[i] + B[i];
```

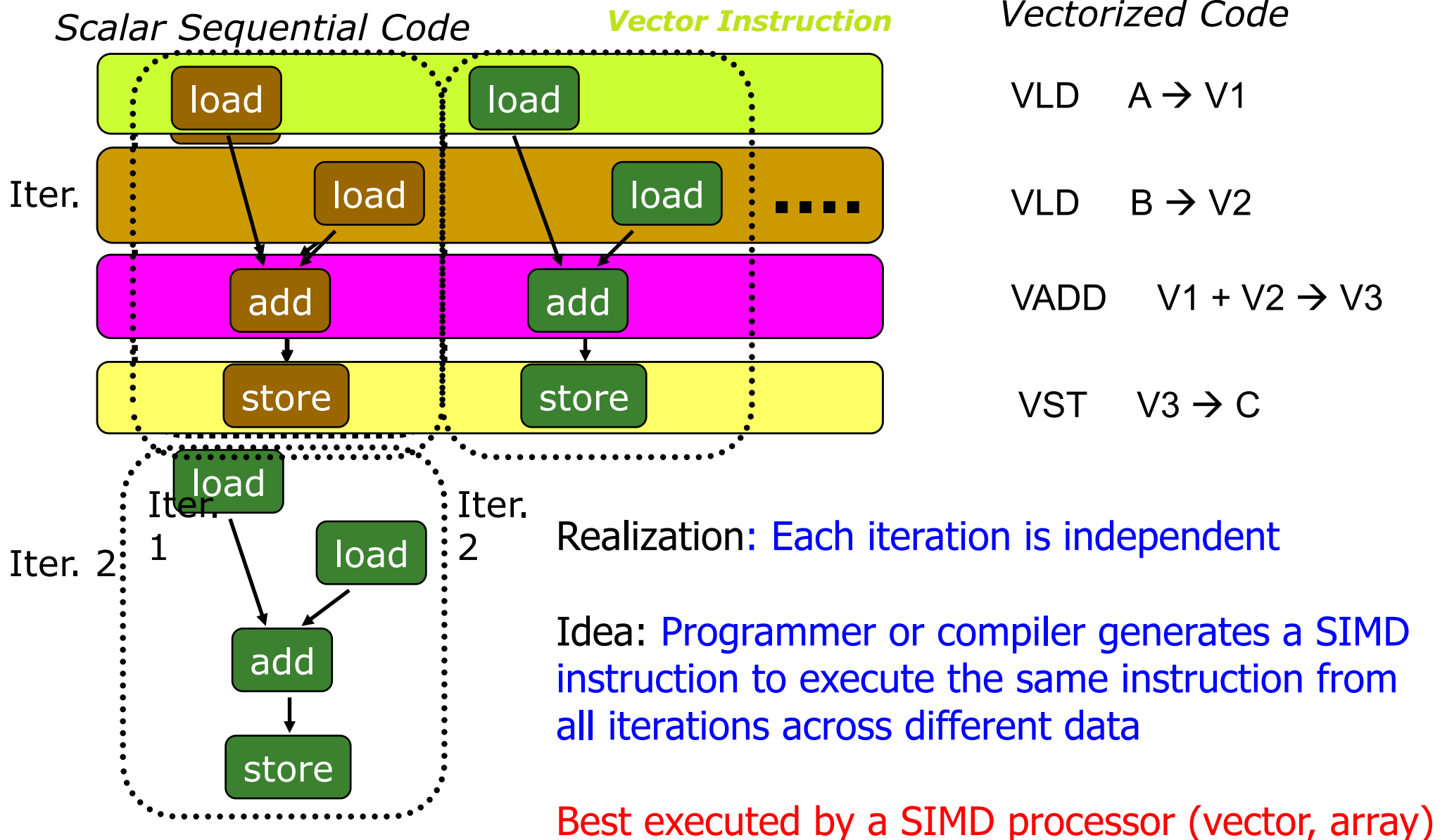
## Scalar Sequential Code



- Can be executed on a:
- Pipelined processor
- Out-of-order execution processor
  - ❑ Independent instructions executed when ready
  - ❑ Different iterations are present in the instruction window and can execute in parallel in multiple functional units
  - ❑ In other words, the loop is dynamically unrolled by the hardware
- Superscalar or VLIW processor
  - ❑ Can fetch and execute multiple instructions per cycle

# Prog. Model 2: Data Parallel (SIMD)

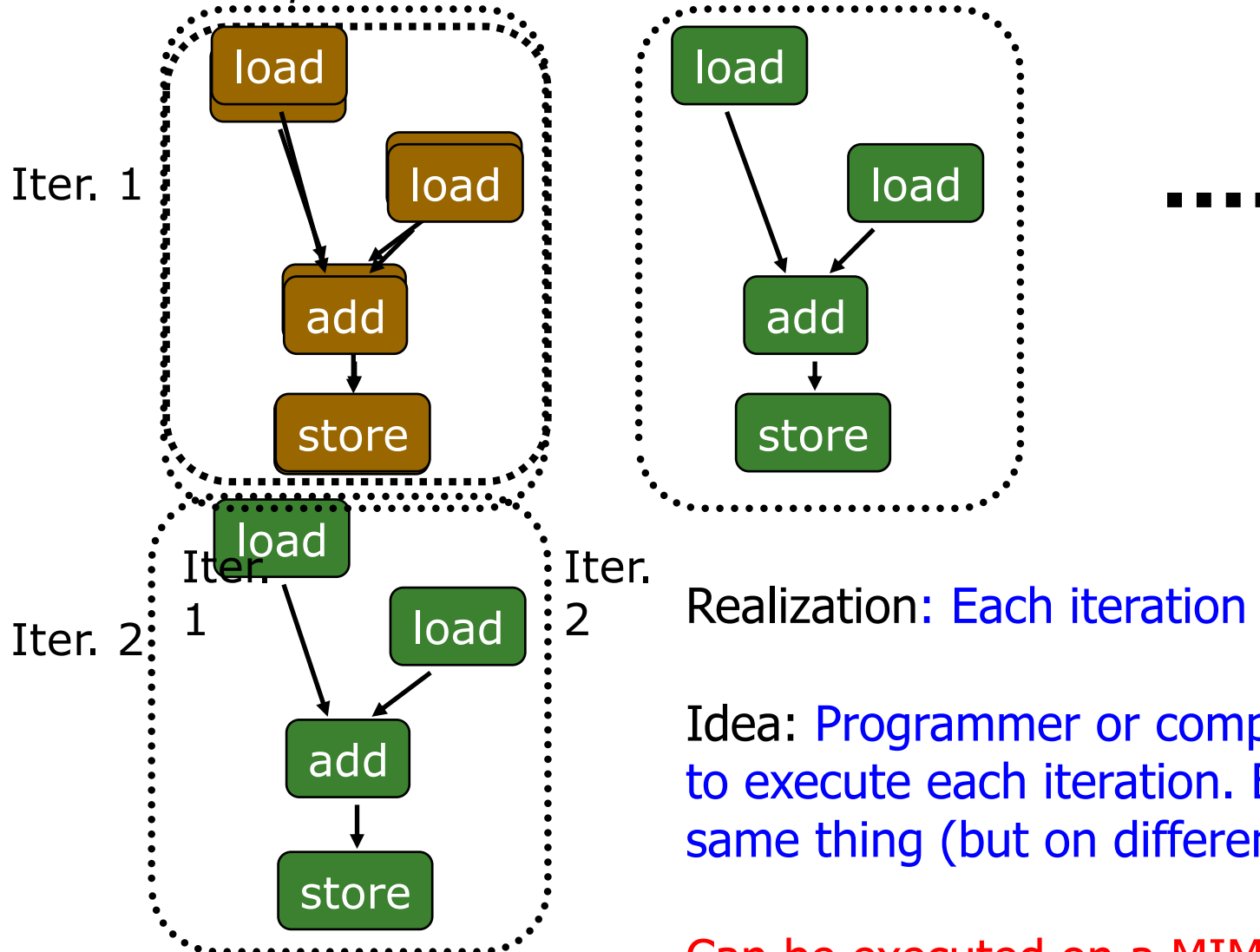
```
for (i=0; i < N; i++)  
  C[i] = A[i] + B[i];
```



# Prog. Model 3: Multithreaded

```
for (i=0; i < N; i++)  
  C[i] = A[i] + B[i];
```

## Scalar Sequential Code



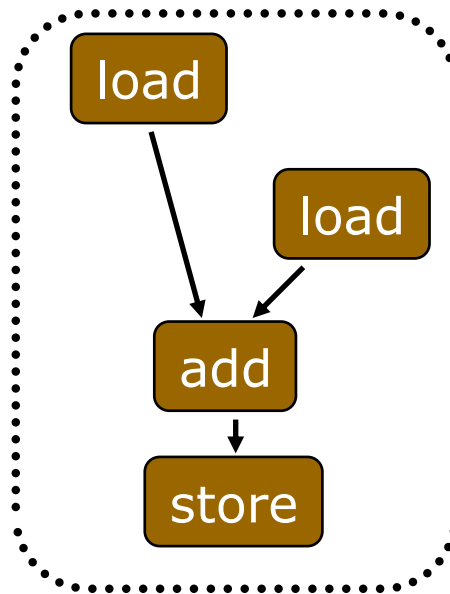
Realization: Each iteration is independent

Idea: Programmer or compiler generates a thread to execute each iteration. Each thread does the same thing (but on different data)

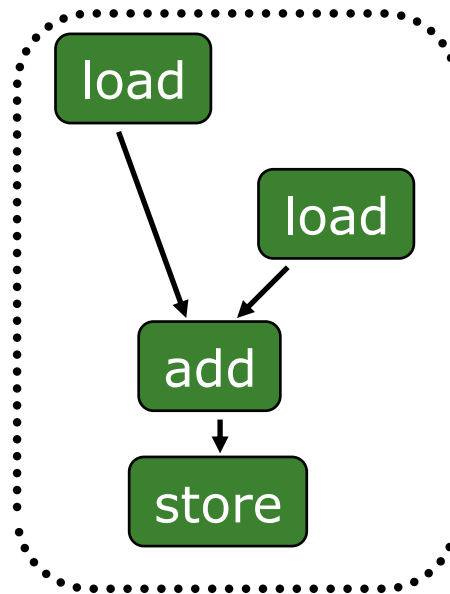
Can be executed on a MIMD machine

# Prog. Model 3: Multithreaded

```
for (i=0; i < N; i++)  
    C[i] = A[i] + B[i];
```



Iter.  
1



Iter.  
2

...

Realization: Each iteration is independent

This particular model is also called:

**SPMD: Single Program Multiple Data**

**Can be executed on a SIMT machine**

**Single Instruction Multiple Thread**

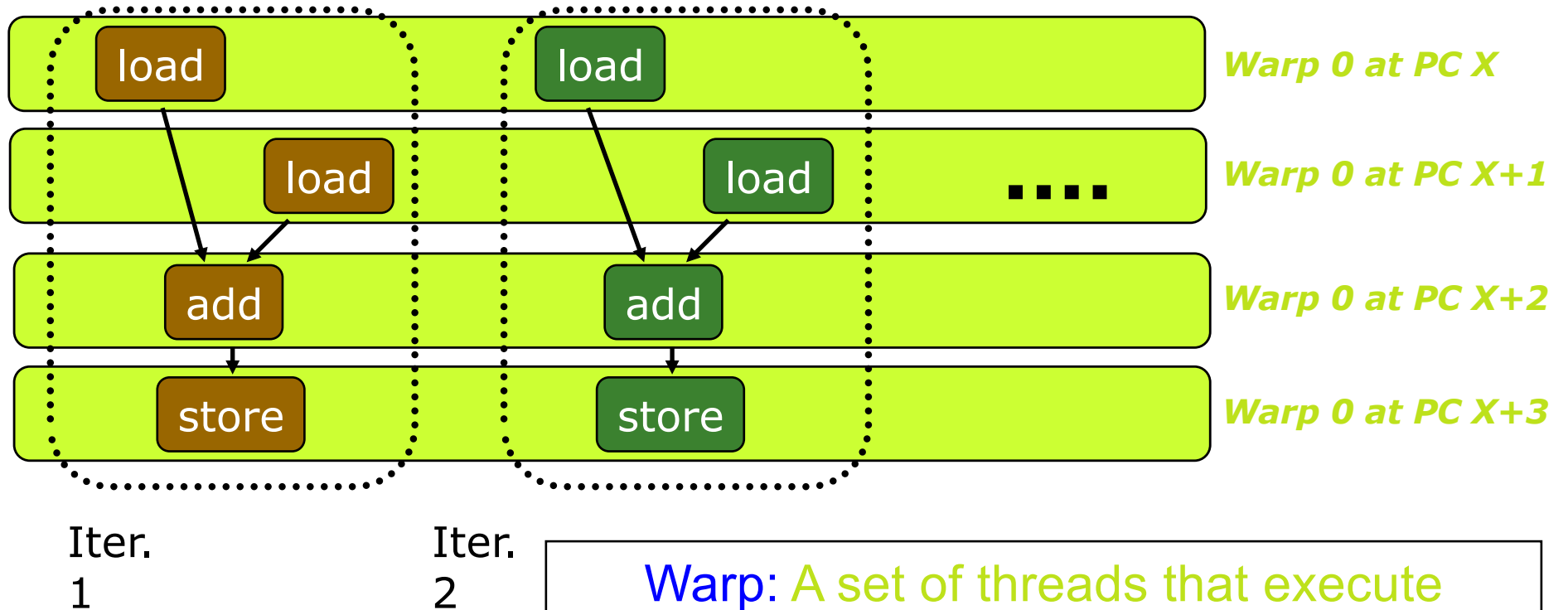
# A GPU is a SIMD (SIMT) Machine

---

- Except it is **not** programmed using SIMD instructions
- It is **programmed using threads** (SPMD programming model)
  - Each thread executes the same code but operates a different piece of data
  - Each thread has its own context (i.e., can be treated/restarted/executed independently)
- A set of threads executing the same instruction are dynamically grouped into a **warp (wavefront)** by the hardware
  - A warp is essentially a **SIMD operation formed by hardware!**

# SPMD on SIMT Machine

```
for (i=0; i < N; i++)  
    C[i] = A[i] + B[i];
```



**Warp:** A set of threads that execute the same instruction (i.e., at the same PC)

This particular model is also called:

**SPMD: Single Program Multiple Data**

**A GPU executes it using the SIMT model:**  
**Single Instruction Multiple Thread**

# Graphics Processing Units

## SIMD not Exposed to Programmer (SIMT)

# SIMD vs. SIMT Execution Model

---

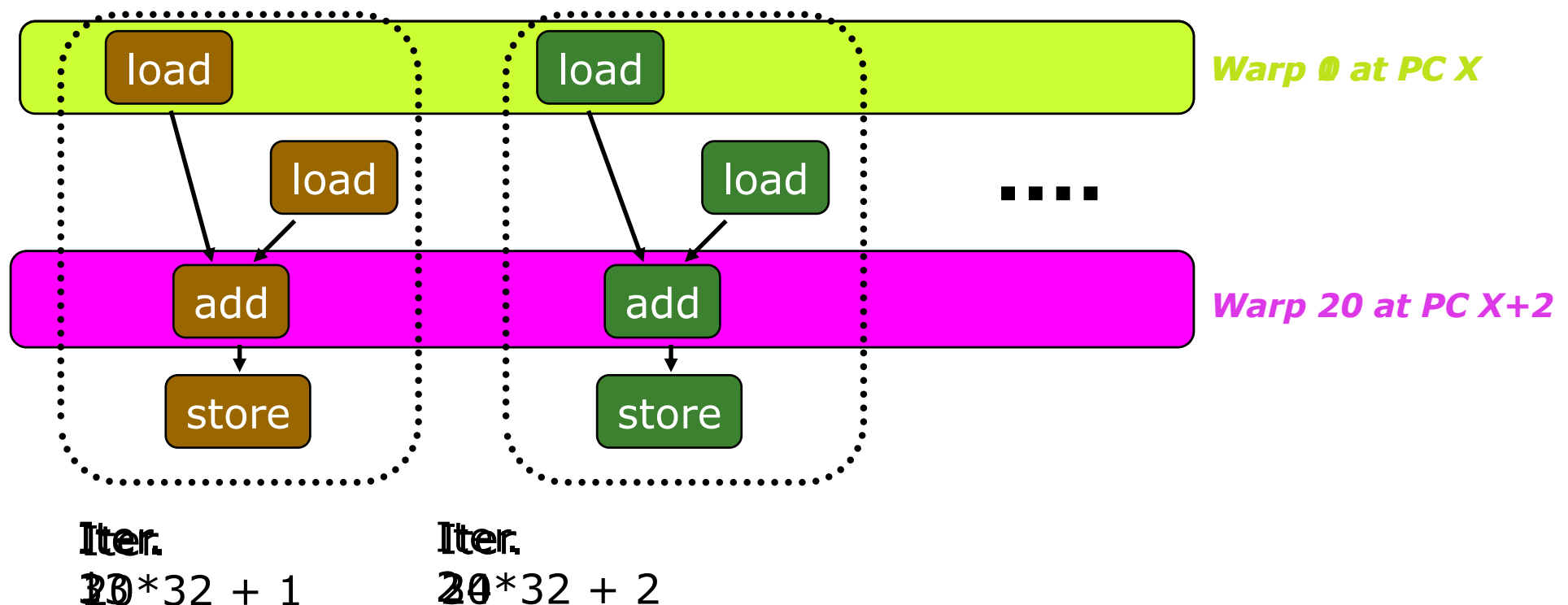
- SIMD: A single **sequential instruction stream** of **SIMD instructions** → each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN
- SIMT: **Multiple instruction streams** of **scalar instructions** → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads
- Two Major SIMT Advantages:
  - **Can treat each thread separately** → i.e., can execute each thread independently (on any type of scalar pipeline) → MIMD processing
  - **Can group threads into warps flexibly** → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of SIMD processing



# Fine-Grained Multithreading of Warps

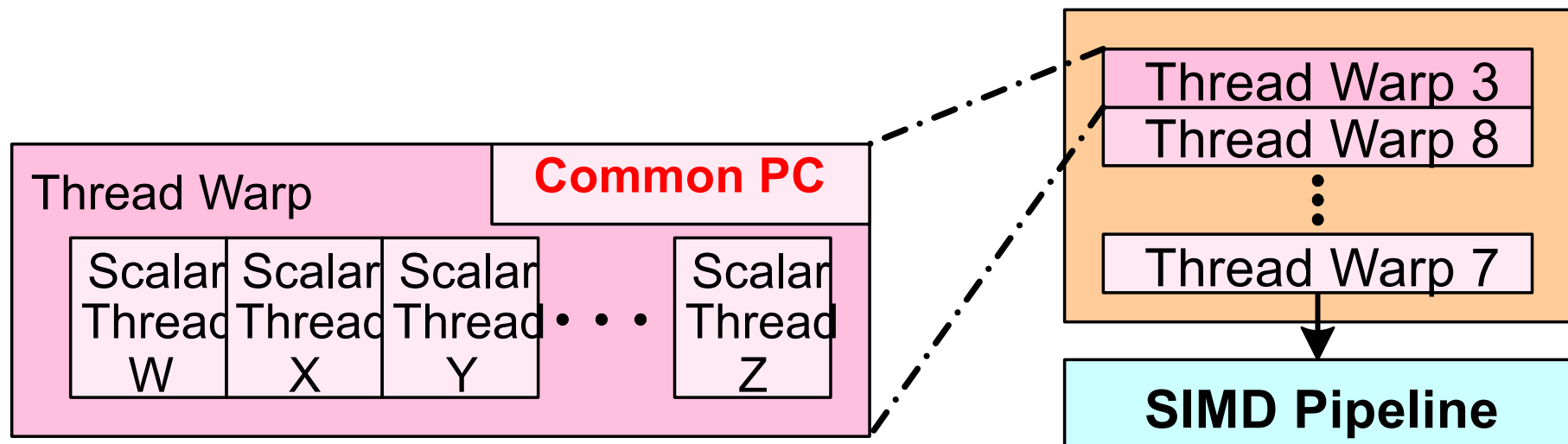
```
for (i=0; i < N; i++)  
    C[i] = A[i] + B[i];
```

- Assume a warp consists of 32 threads
- If you have 32K iterations, and 1 iteration/thread  $\rightarrow$  1K warps
- Warps can be interleaved on the same pipeline  $\rightarrow$  Fine grained multithreading of warps

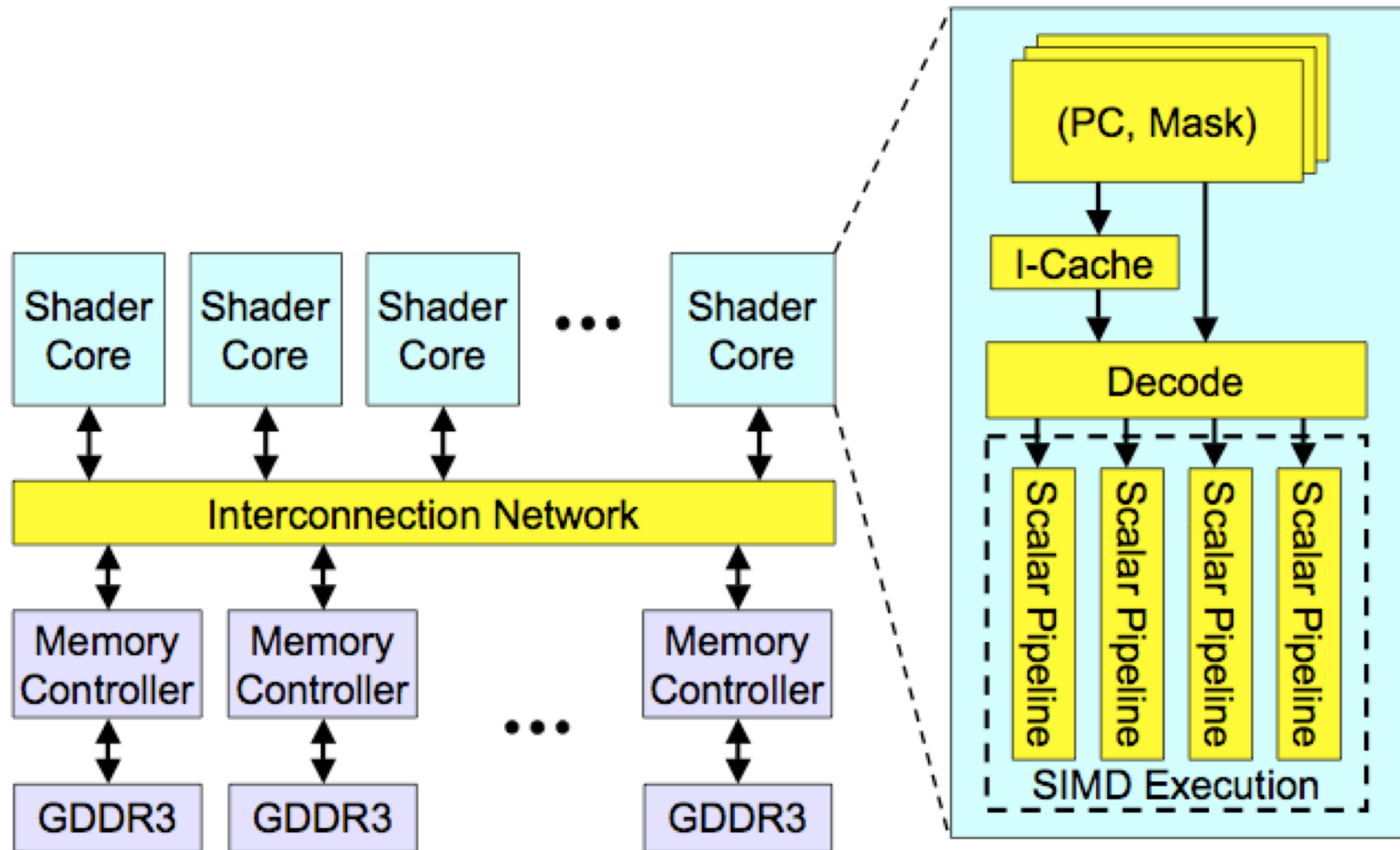


# Warps and Warp-Level FGMT

- Warp: A set of threads that execute the same instruction (on different data elements) → SIMT (Nvidia-speak)
- All threads run the same code
- Warp: The threads that run lengthwise in a woven fabric ...

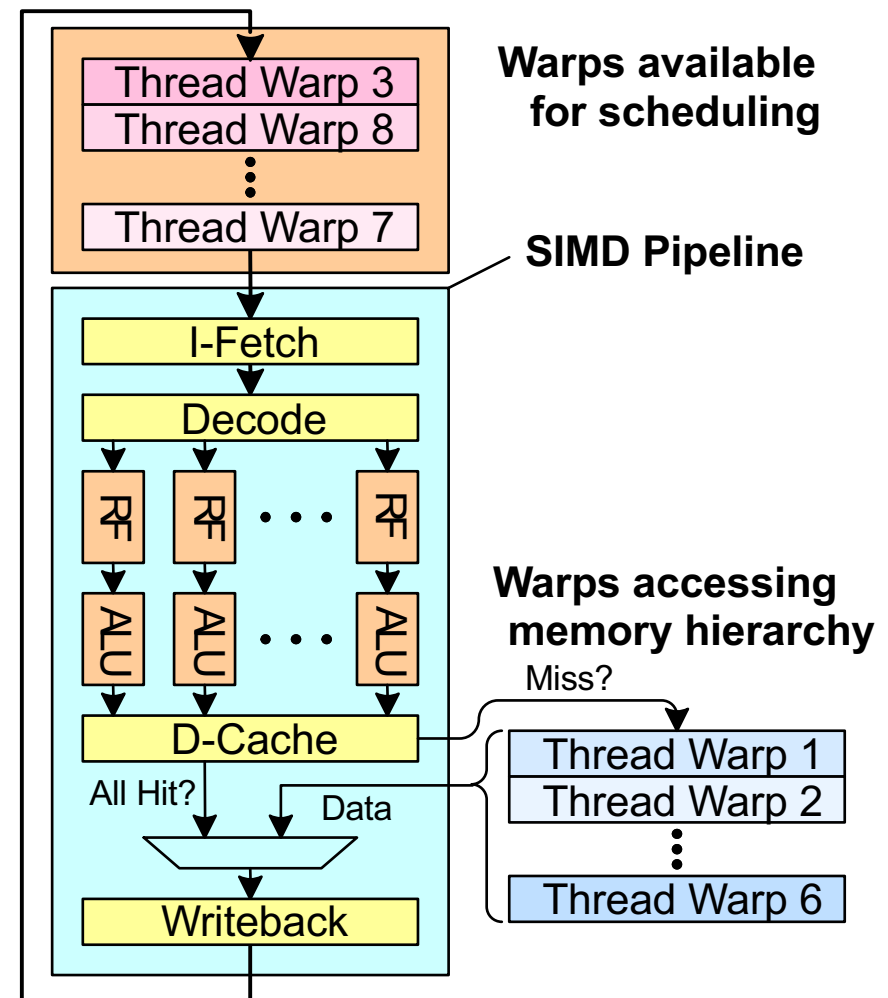


# High-Level View of a GPU



# Latency Hiding via Warp-Level FGMT

- Warp: A set of threads that execute the same instruction (on different data elements)
- Fine-grained multithreading
  - One instruction per thread in pipeline at a time (No interlocking)
  - Interleave warp execution to hide latencies
- Register values of all threads stay in register file
- FGMT enables long latency tolerance
  - Millions of pixels

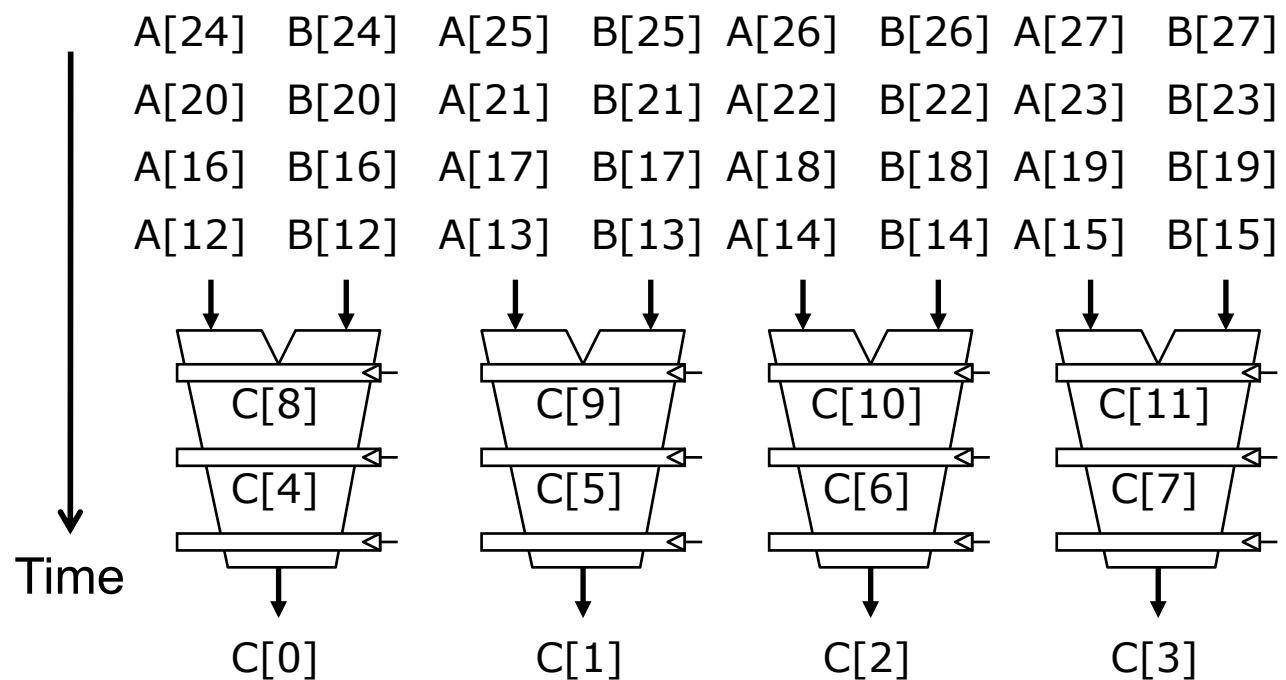
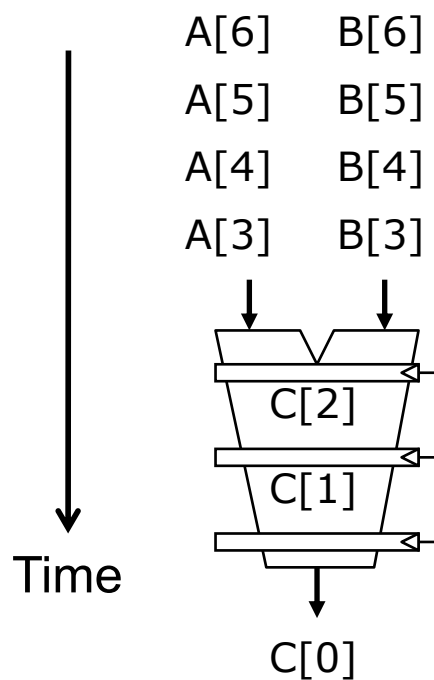


# Warp Execution (Recall the Slide)

32-thread warp executing  $\text{ADD } A[\text{tid}], B[\text{tid}] \rightarrow C[\text{tid}]$

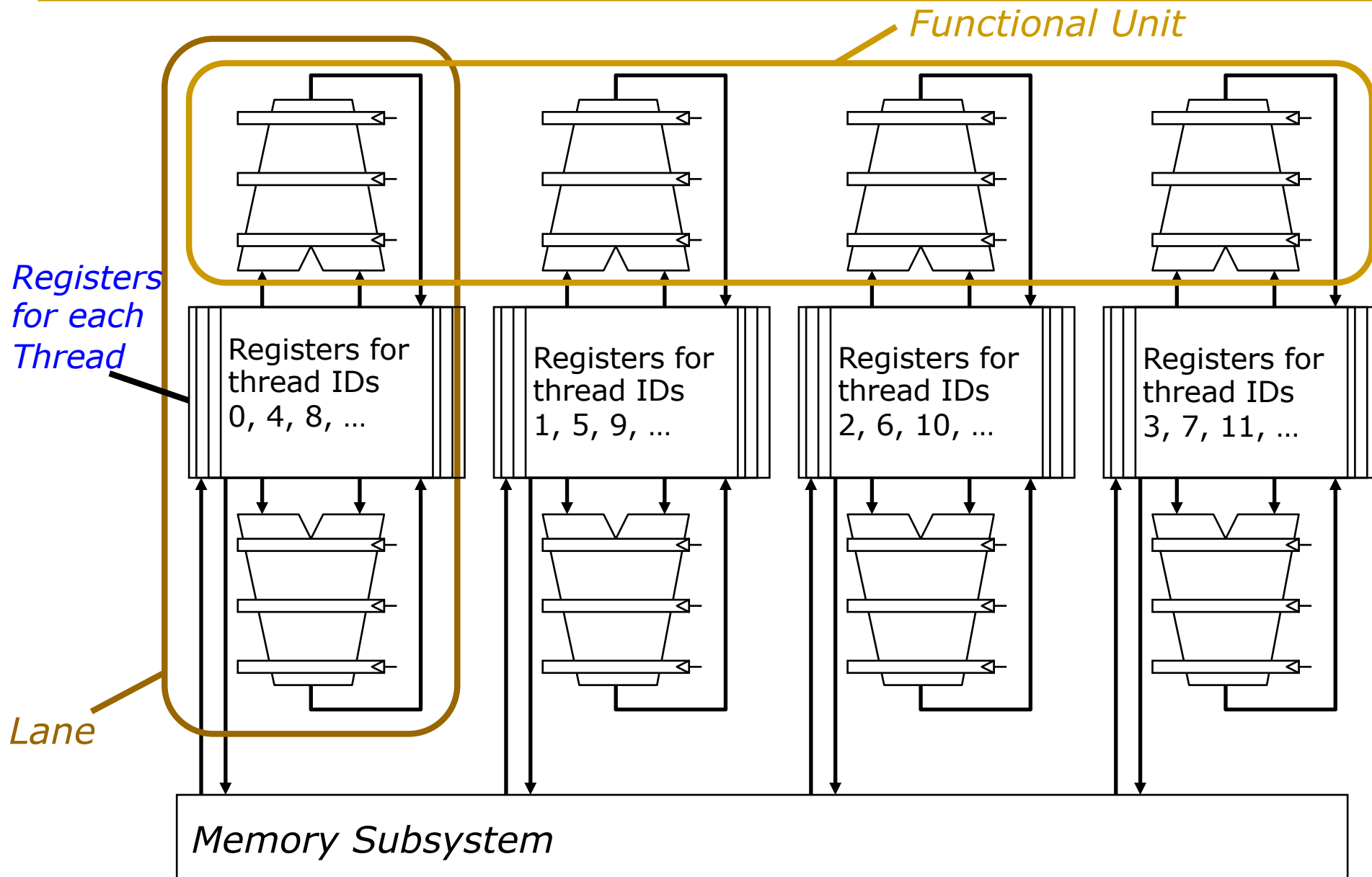
*Execution using  
one pipelined  
functional unit*

*Execution using  
four pipelined  
functional units*



← Space →

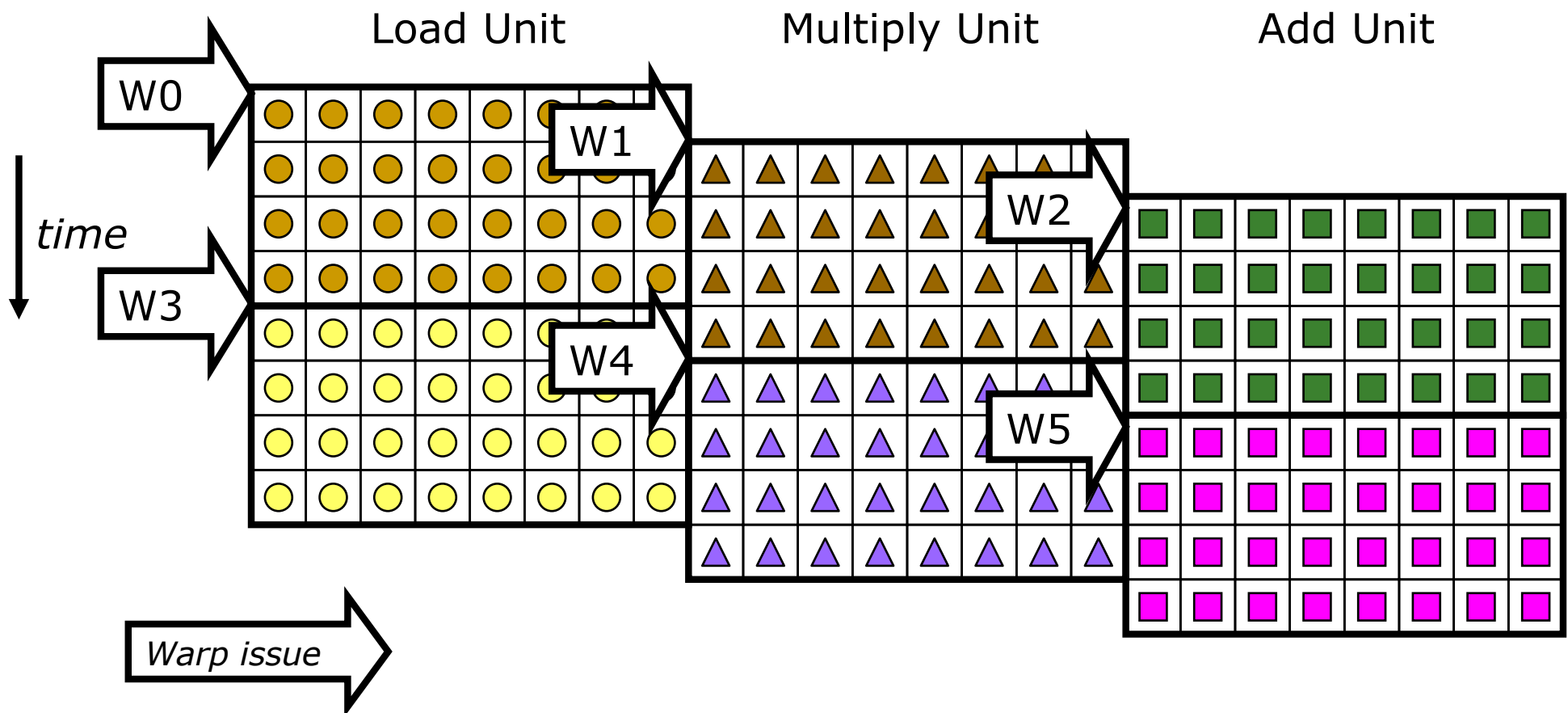
# SIMD Execution Unit Structure



# Warp Instruction Level Parallelism

Can overlap execution of multiple instructions

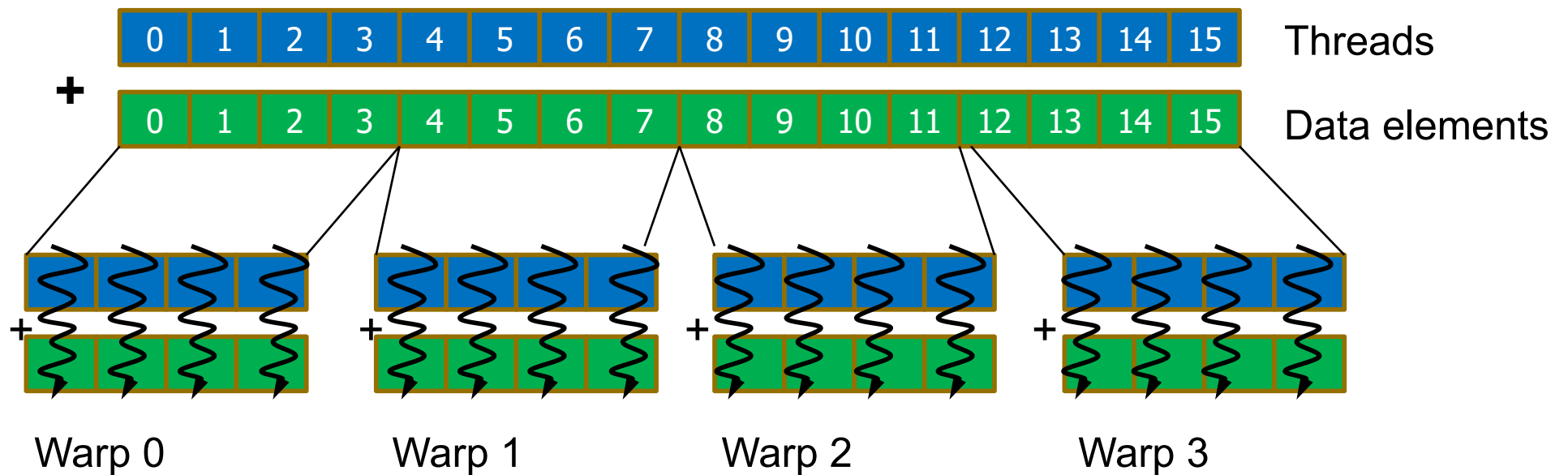
- Example machine has 32 threads per warp and 8 lanes
- Completes 24 operations/cycle while issuing 1 warp/cycle



# SIMT Memory Access

- Same instruction in different threads uses **thread id** to index and access different data elements

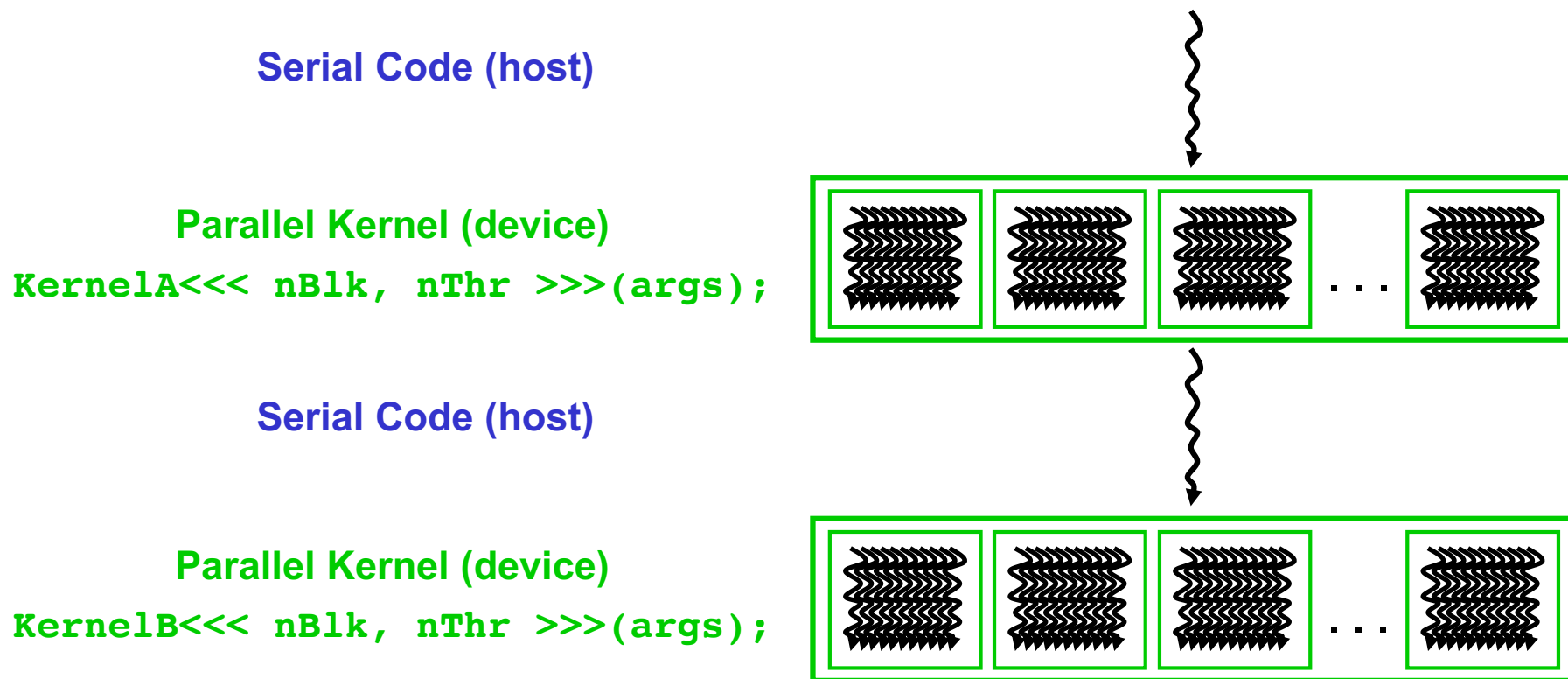
Let's assume  $N=16$ , 4 threads per warp  $\rightarrow$  4 warps





# Warps *not* Exposed to GPU Programmers

- CPU threads and GPU kernels
  - Sequential or modestly parallel sections on CPU
  - Massively parallel sections on GPU: Blocks of threads



Slide credit: Hwu & Kirk

# Sample GPU SIMT Code (Simplified)

---

CPU code

```
for (ii = 0; ii < 100000; ++ii) {  
    C[ii] = A[ii] + B[ii];  
}
```



CUDA code

```
// there are 100000 threads  
__global__ void KernelFunction(...) {  
    int tid = blockDim.x * blockIdx.x + threadIdx.x;  
    int varA = aa[tid];  
    int varB = bb[tid];  
    C[tid] = varA + varB;  
}
```

# Sample GPU Program (Less Simplified)

## CPU Program

```
void add_matrix
( float *a, float* b, float *c, int N) {
    int index;
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j) {
            index = i + j*N;
            c[index] = a[index] + b[index];
        }
}

int main () {
    add_matrix (a, b, c, N);
}
```

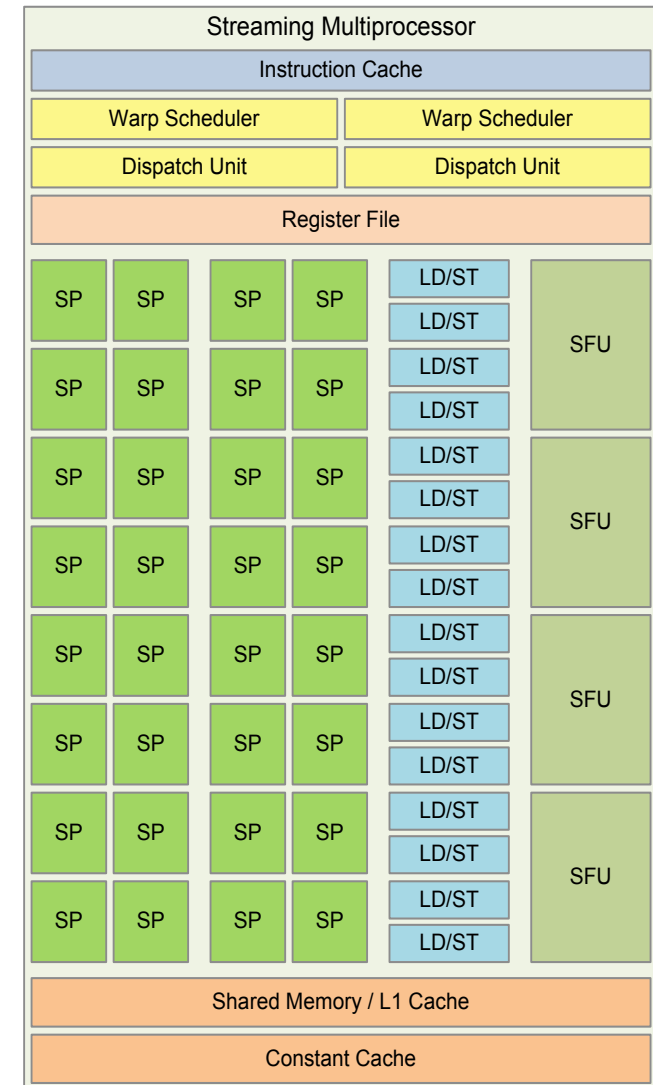
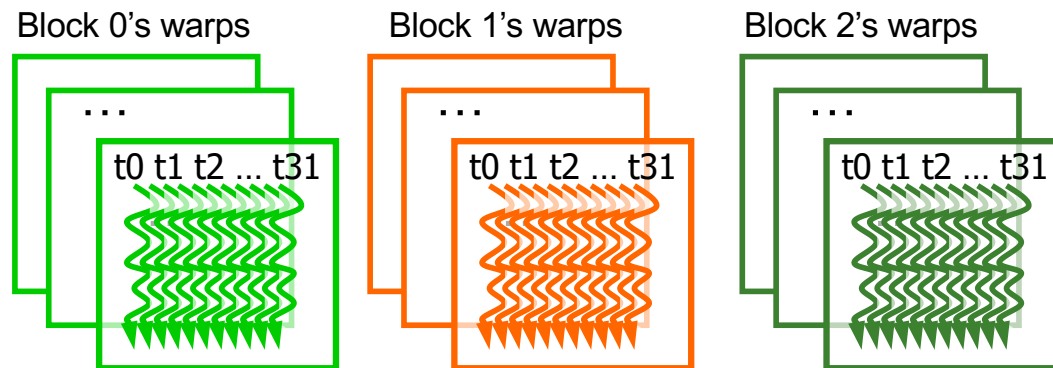
## GPU Program

```
__global__ add_matrix
( float *a, float *b, float *c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;
    if (i < N && j < N)
        c[index] = a[index]+b[index];
}

int main() {
    dim3 dimBlock( blocksize, blocksize) ;
    dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
    add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N);
}
```

# From Blocks to Warps

- GPU cores: SIMD pipelines
  - ❑ Streaming Multiprocessors (SM)
  - ❑ Streaming Processors (SP)
- Blocks are divided into **warps**
  - ❑ SIMD unit (32 threads)



NVIDIA Fermi architecture

# Warp-based SIMD vs. Traditional SIMD

---

- Traditional **SIMD** contains a single thread
  - Sequential instruction execution; lock-step operations in a SIMD instruction
  - Programming model is SIMD (no extra threads) → SW needs to know vector length
  - ISA contains vector/SIMD instructions
- Warp-based SIMD consists of multiple scalar threads executing in a SIMD manner (i.e., same instruction executed by all threads)
  - Does not have to be lock step
  - Each thread can be treated individually (i.e., placed in a different warp)  
→ programming model not SIMD
    - SW does not need to know vector length
    - Enables multithreading and flexible dynamic grouping of threads
  - ISA is scalar → SIMD operations can be formed dynamically
  - Essentially, it is SPMD programming model implemented on SIMD hardware

# SPMD

---

- Single procedure/program, multiple data
  - This is a programming model rather than computer organization
- Each processing element executes the same procedure, except on different data elements
  - Procedures can synchronize at certain points in program, e.g. barriers
- Essentially, multiple instruction streams execute the same program
  - Each program/procedure 1) works on different data, 2) can execute a different control-flow path, at run-time
  - Many scientific applications are programmed this way and run on MIMD hardware (multiprocessors)
  - Modern GPUs programmed in a similar way on a SIMD hardware

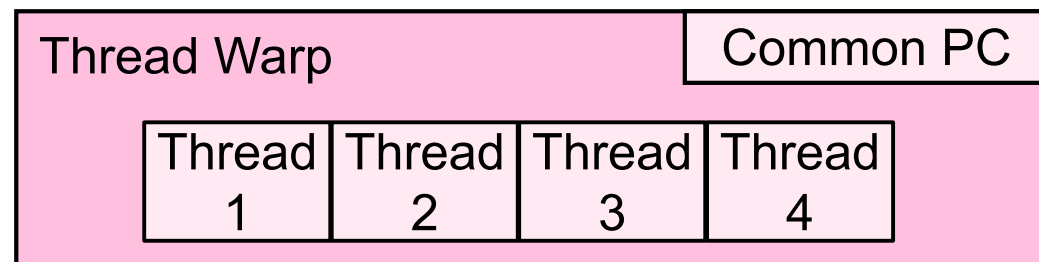
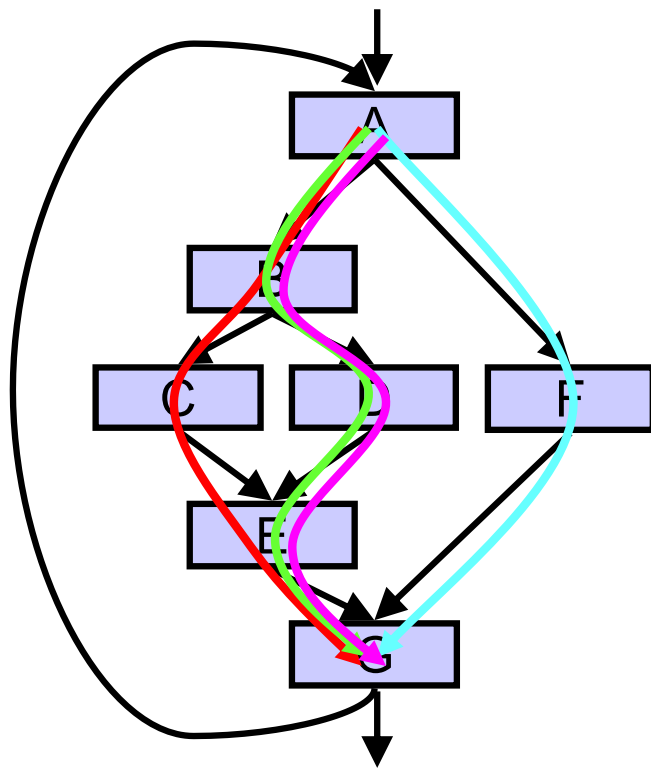
# SIMD vs. SIMT Execution Model

---

- SIMD: A single **sequential instruction stream** of **SIMD instructions** → each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN
- SIMT: **Multiple instruction streams** of **scalar instructions** → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads
- Two Major SIMT Advantages:
  - **Can treat each thread separately** → i.e., can execute each thread independently on any type of scalar pipeline → MIMD processing
  - **Can group threads into warps flexibly** → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of SIMD processing

# Threads Can Take Different Paths in Warp-based SIMD

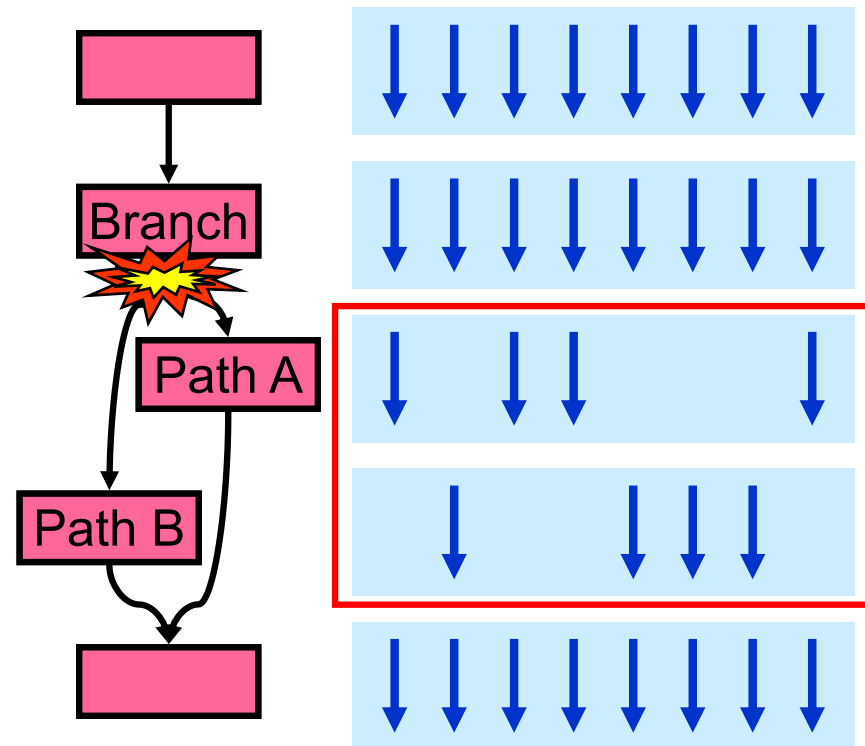
- Each thread can have **conditional control flow instructions**
- Threads can execute different control flow paths





# Control Flow Problem in GPUs/SIMT

- A GPU uses a SIMD pipeline to save area on control logic
  - Groups scalar threads into warps
- **Branch divergence** occurs when threads inside warps branch to different execution paths



**This is the same as conditional/predicated/masked execution.  
Recall the Vector Mask and Masked Vector Operations?**

# Remember: Each Thread Is Independent

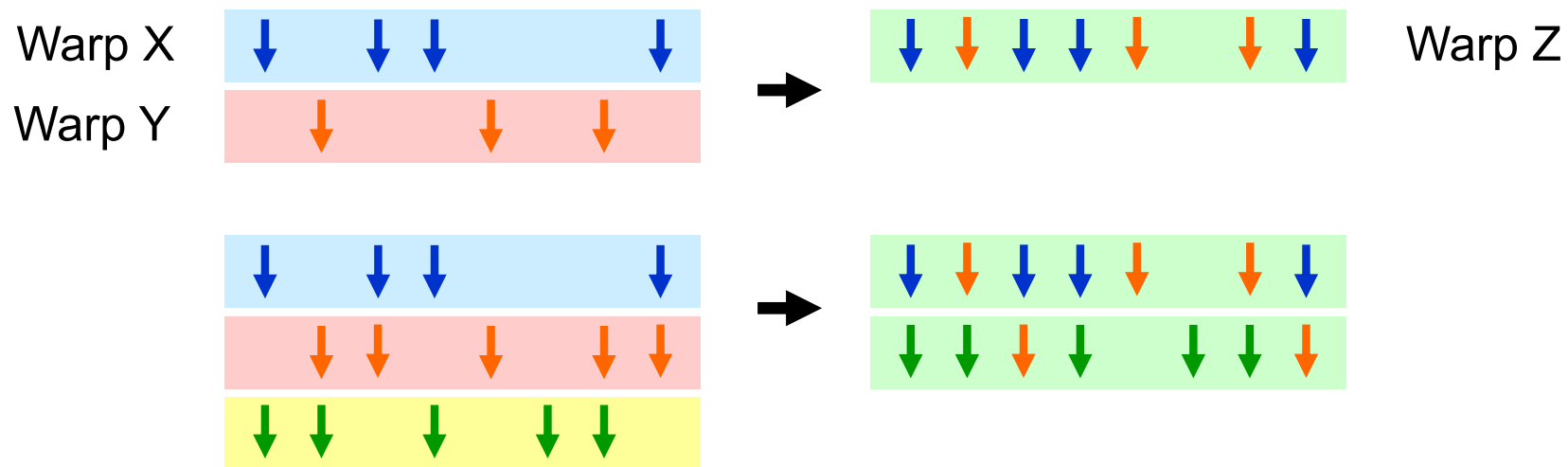
---

- Two Major SIMT Advantages:
  - Can treat each thread separately → i.e., can execute each thread independently on any type of scalar pipeline → MIMD processing
  - Can group threads into warps flexibly → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of SIMD processing
- If we have many threads
- We can find individual threads that are at the same PC
- And, group them together into a single warp dynamically
- This reduces “divergence” → improves SIMD utilization
  - SIMD utilization: fraction of SIMD lanes executing a useful operation (i.e., executing an active thread)

# Dynamic Warp Formation/Merging

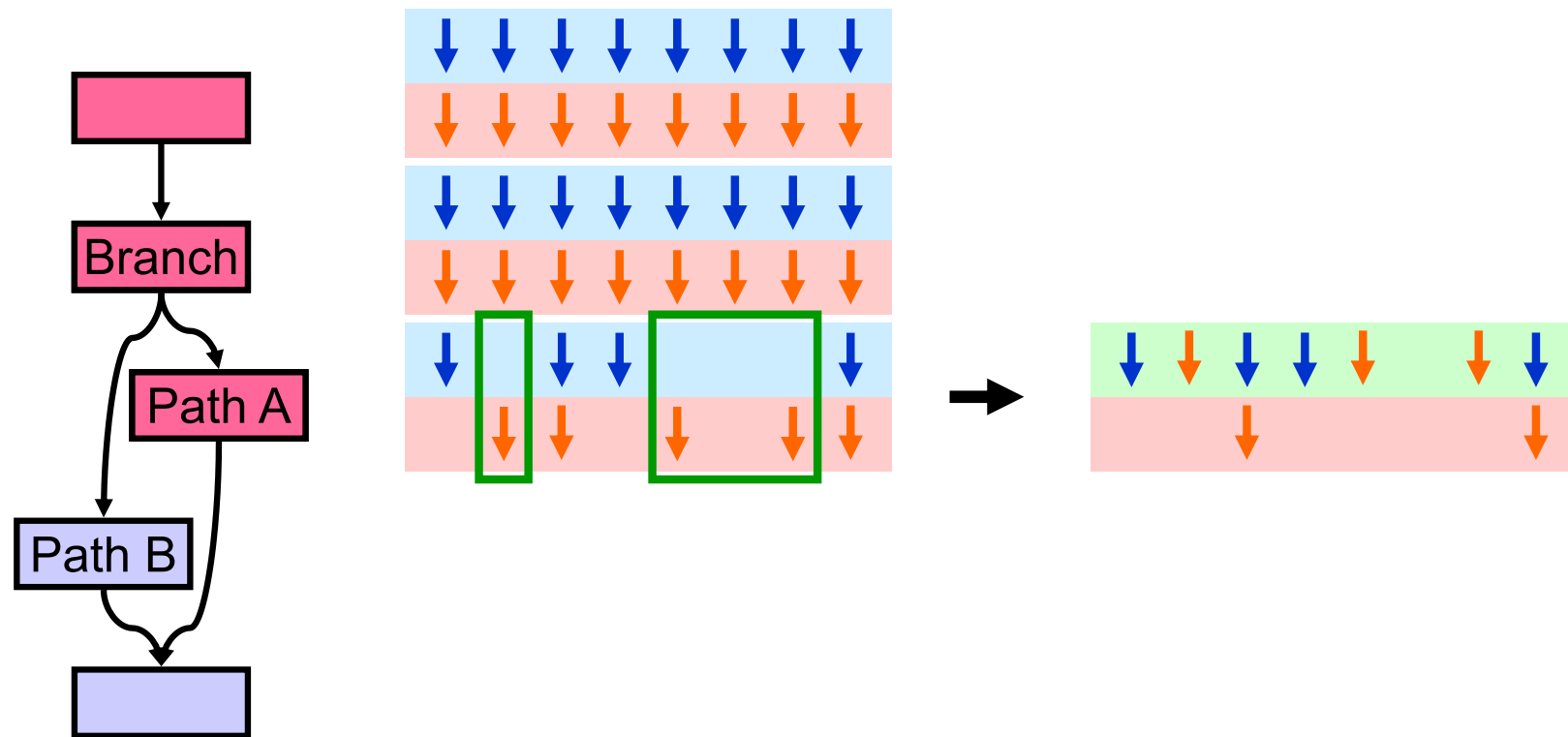
---

- Idea: Dynamically merge threads executing the same instruction (after branch divergence)
- Form new warps from warps that are waiting
  - Enough threads branching to each path enables the creation of full new warps



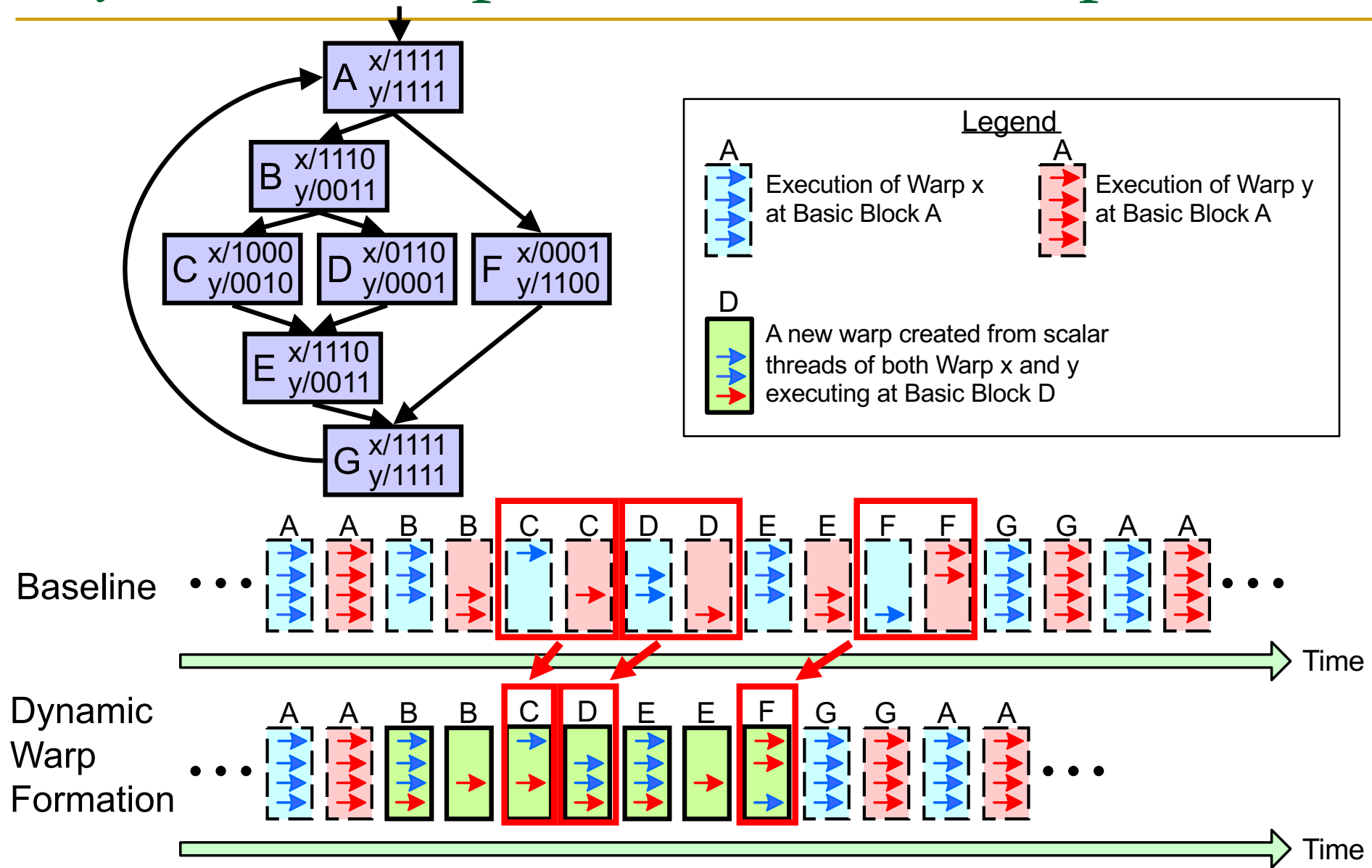
# Dynamic Warp Formation/Merging

- Idea: Dynamically merge threads executing the same instruction (after branch divergence)

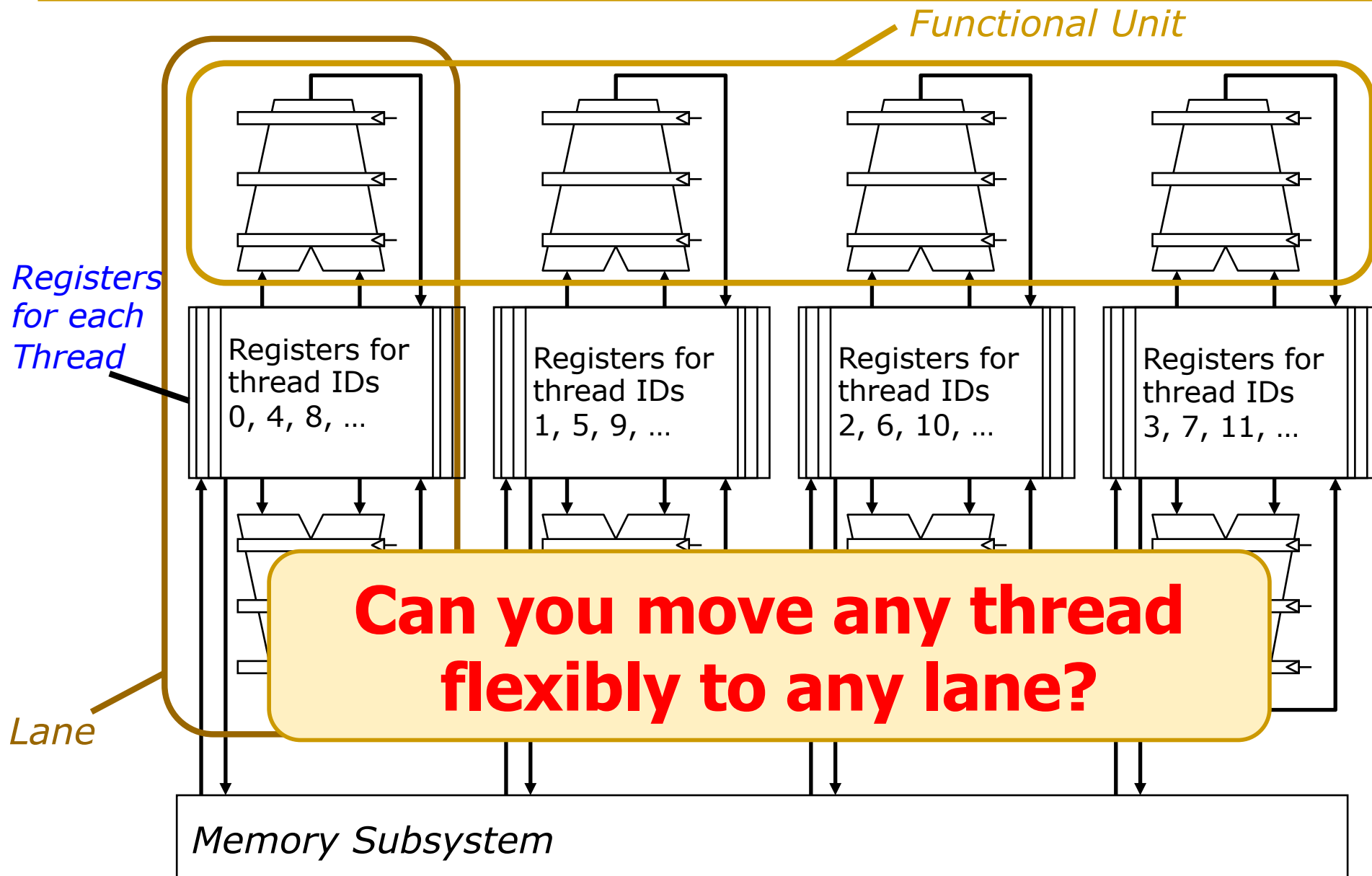


- Fung et al., “Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow,” MICRO 2007.

# Dynamic Warp Formation Example

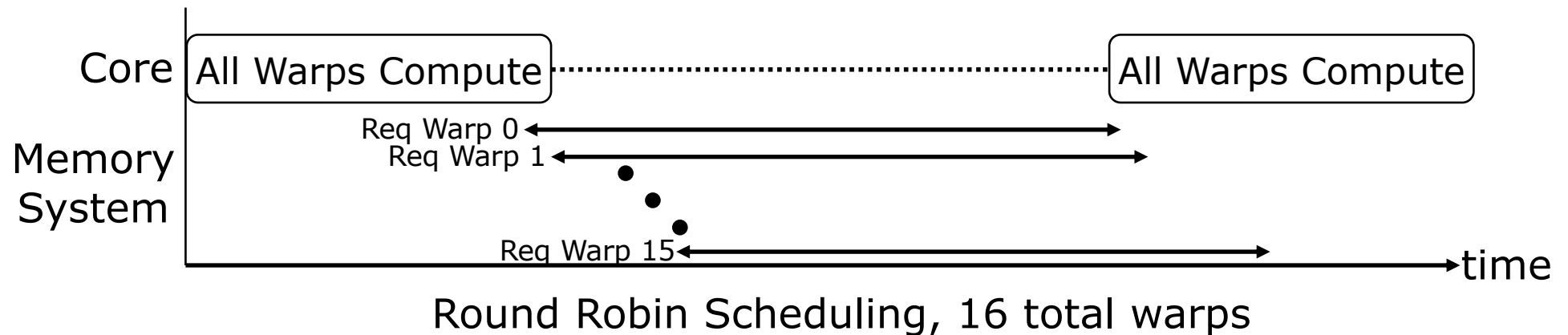


# Hardware Constraints Limit Flexibility of Warp Grouping



# Large Warps and Two-Level Warp Scheduling

- Two main reasons for GPU resources be underutilized
  - Branch divergence
  - Long latency operations



# Large Warp Microarchitecture Example

- Reduce **branch divergence** by having large warps
- **Dynamically break down a large warp into sub-warps**

Decode Stage

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Sub-warp 0 mask

1	1	1	1
---	---	---	---

Sub-warp 0 mask

1	1	1	1
---	---	---	---

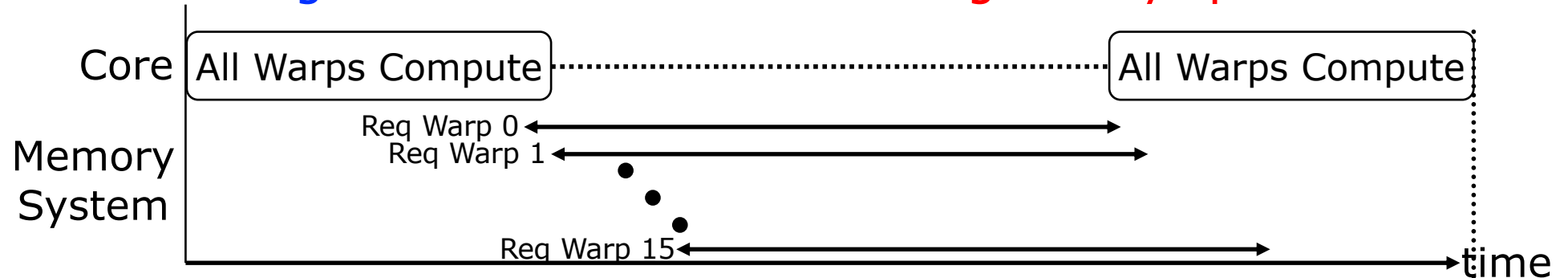
Sub-warp 0 mask

1	1	1	1
---	---	---	---

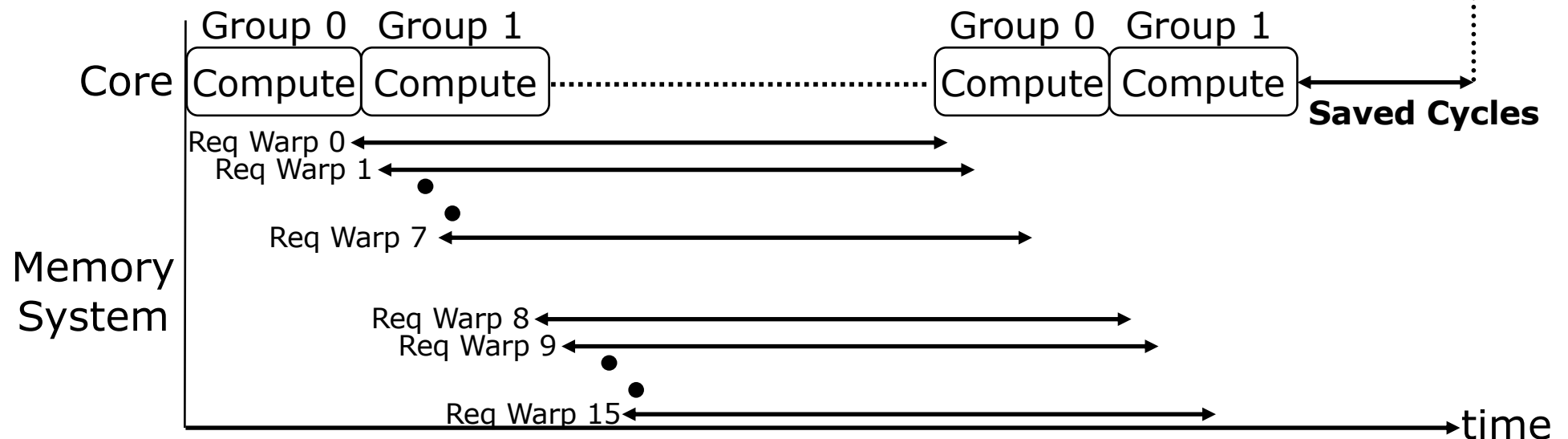


# Two-Level Round Robin

- Scheduling in two levels to deal with long latency operations



Round Robin Scheduling, 16 total warps



Two Level Round Robin Scheduling, 2 fetch groups, 8 warps each

# An Example GPU

# NVIDIA GeForce GTX 285

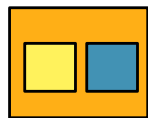
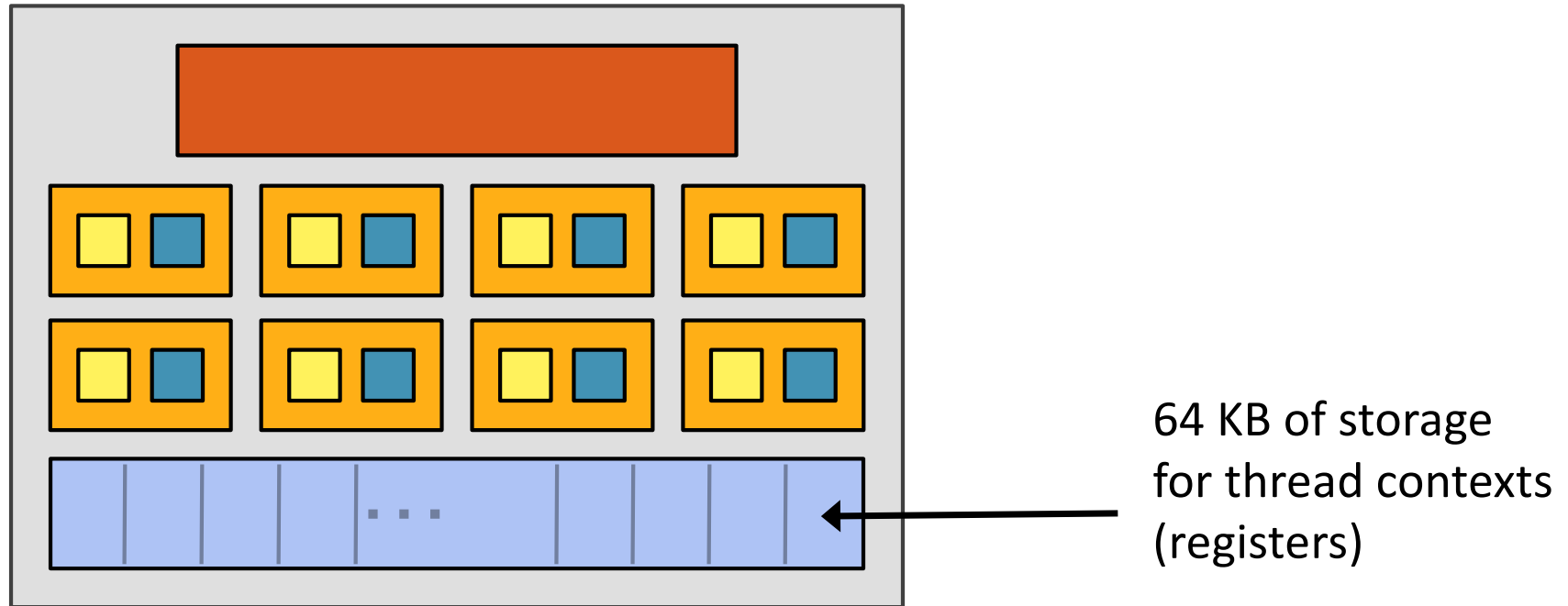
---

- NVIDIA-speak:
  - ❑ 240 stream processors
  - ❑ “SIMT execution”
- Generic speak:
  - ❑ 30 cores
  - ❑ 8 SIMD functional units per core



# NVIDIA GeForce GTX 285 “core”

---



= SIMD functional unit, control shared across 8 units



= multiply-add



= multiply



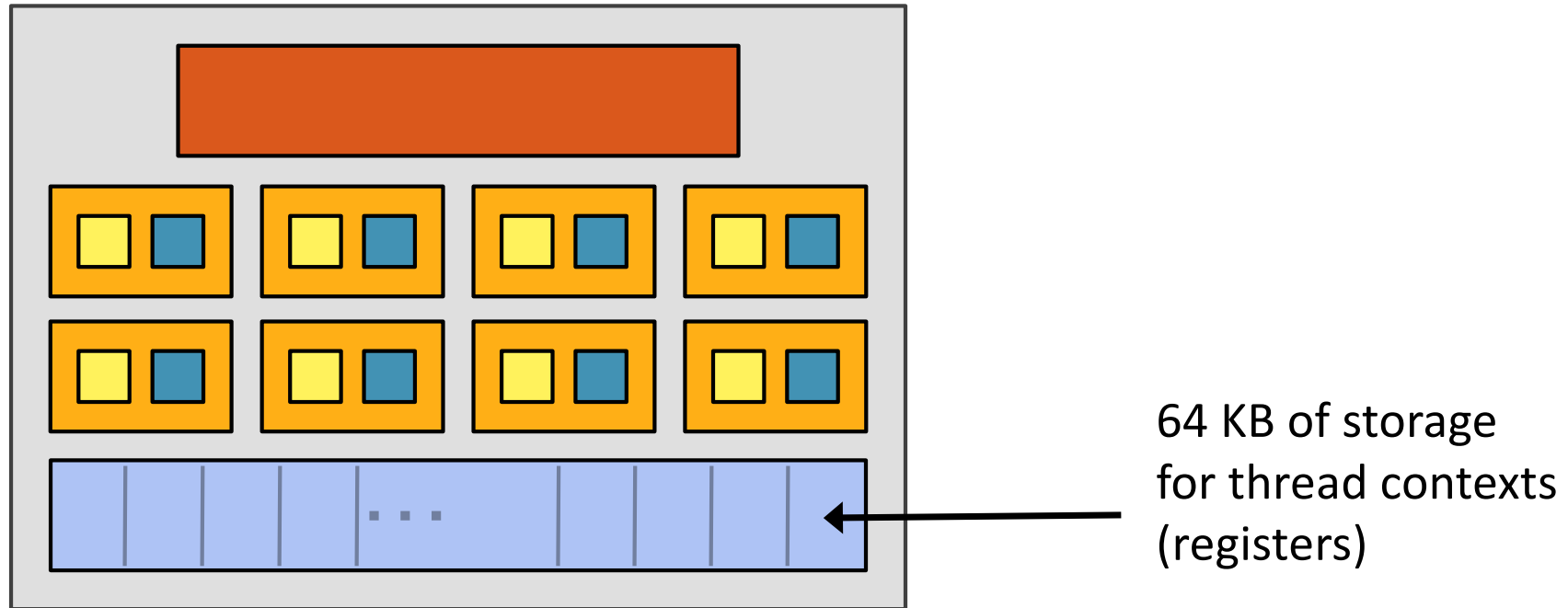
= instruction stream decode



= execution context storage

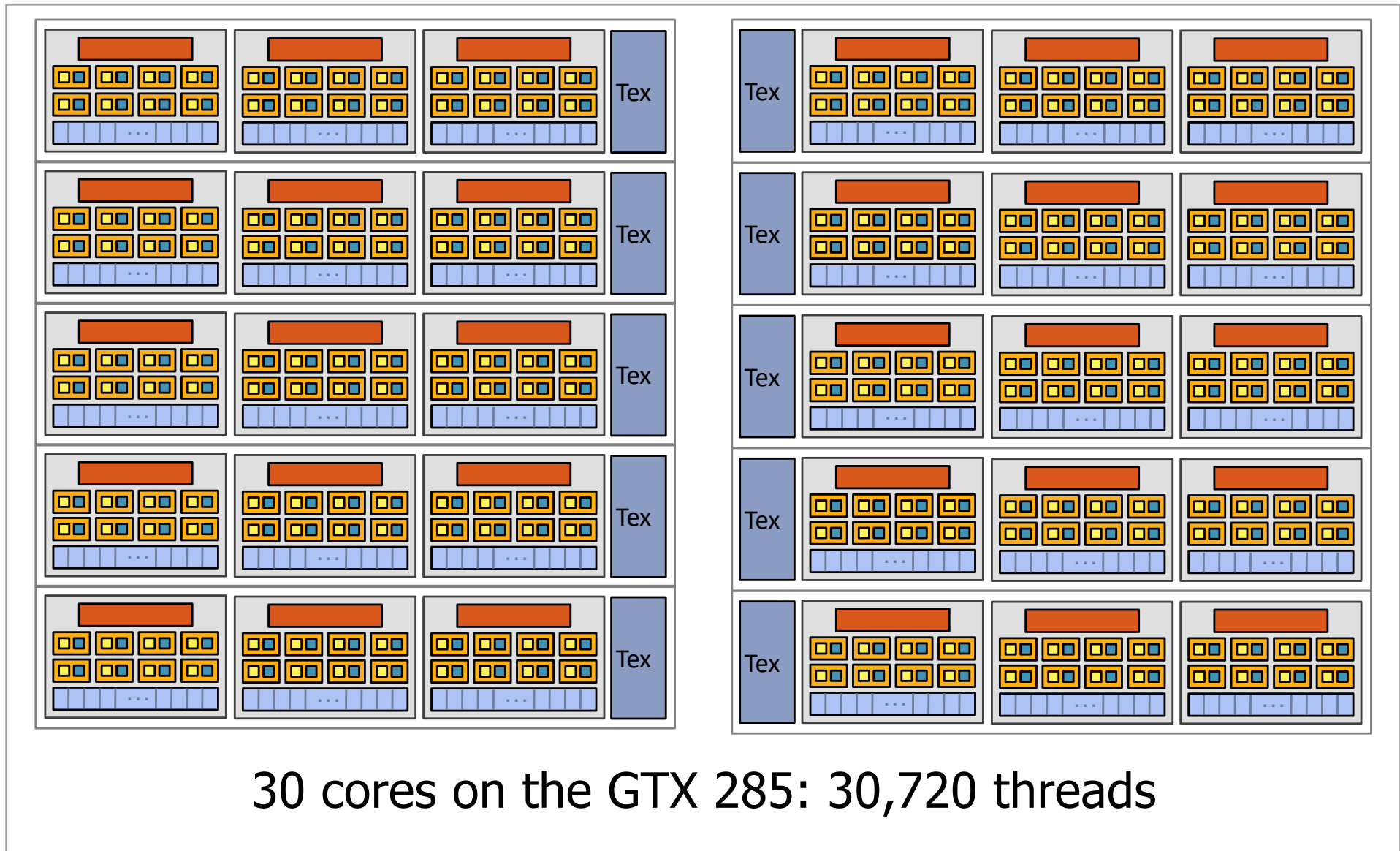
# NVIDIA GeForce GTX 285 “core”

---

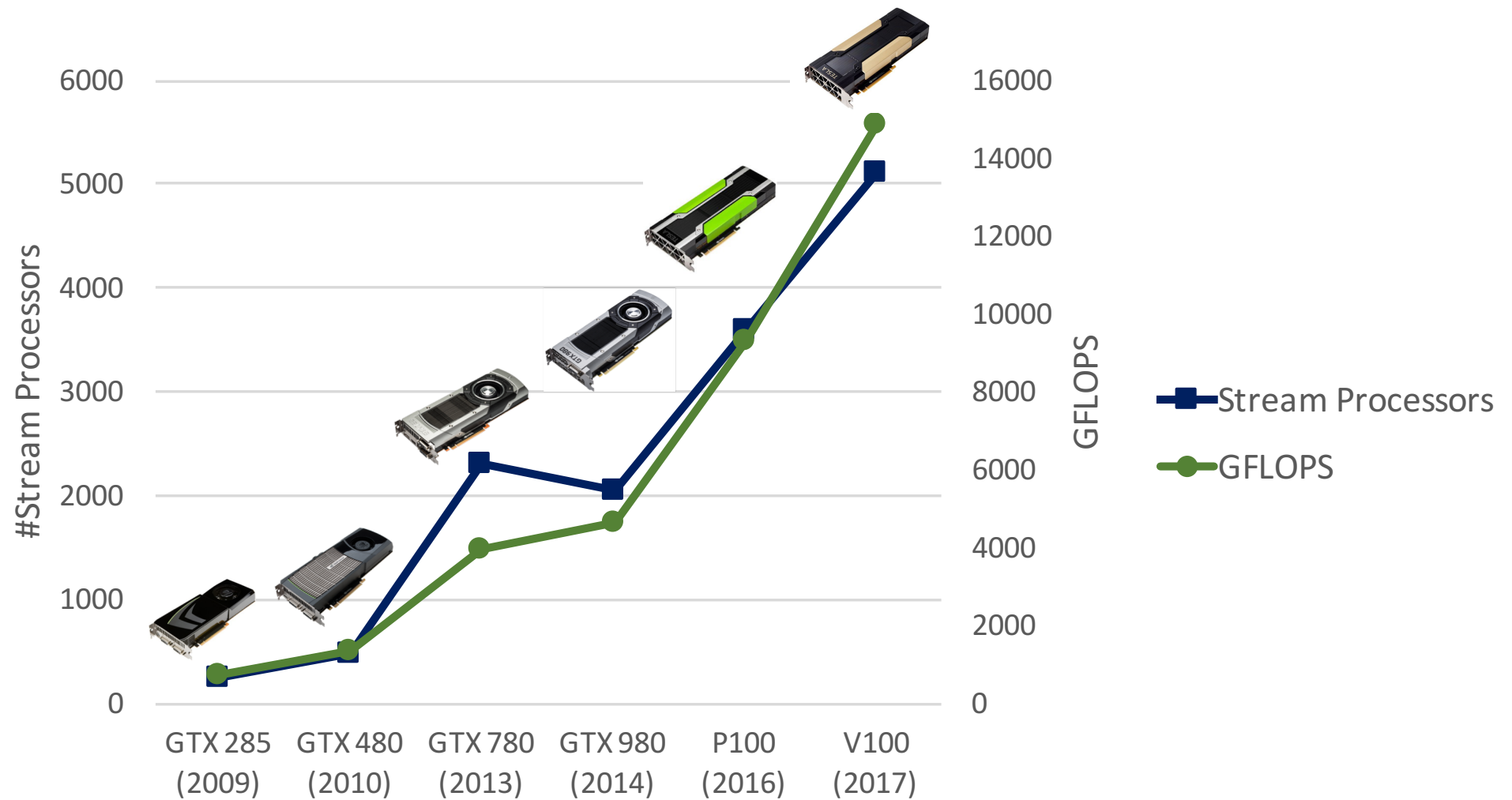


- Groups of 32 **threads** share instruction stream (each group is a Warp)
- Up to 32 warps are simultaneously interleaved
- Up to 1024 thread contexts can be stored

# NVIDIA GeForce GTX 285



# Evolution of NVIDIA GPUs



# NVIDIA V100

---

- NVIDIA-speak:
  - ❑ 5120 stream processors
  - ❑ “SIMT execution”
- Generic speak:
  - ❑ 80 cores
  - ❑ 64 SIMD functional units per core
  - ❑ Tensor cores for Machine Learning
- NVIDIA, “[NVIDIA Tesla V100 GPU Architecture. White Paper,](#)” 2017.





# NVIDIA V100 Block Diagram



<https://devblogs.nvidia.com/inside-volta/>

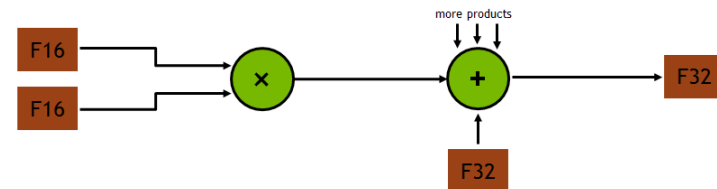
80 cores on the V100

# NVIDIA V100 Core



15.7 TFLOPS Single Precision  
7.8 TFLOPS Double Precision  
125 TFLOPS for Deep Learning (Tensor cores)

FP16 storage/input      Full precision product      Sum with FP32 accumulator      Convert to FP32 result



$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32      FP16      FP16      FP16 or FP32

<https://devblogs.nvidia.com/inside-volta/>

# Food for Thought

---

- Compare and contrast **GPUs** vs **Systolic Arrays**
  - Which one is better for machine learning?
  - Which one is better for image/vision processing?
  - What types of parallelism each one exploits?
  - What are the tradeoffs?
- If you are interested in such questions and more...
  - **Bachelor's Seminar in Computer Architecture** (HS2019, FS2020)
  - **Computer Architecture Master's Course** (HS2019)

# Design of Digital Circuits

## Lecture 21: Graphics Processing Units

Prof. Onur Mutlu

ETH Zurich

Spring 2019

10 May 2019